João Pedro Matos da Costa

MASSIVELY SCALABLE DATA WAREHOUSES WITH PERFORMANCE PREDICTABILITY

Universidade de Coimbra

João Pedro Matos da Costa

MASSIVELY SCALABLE DATA WAREHOUSES WITH PERFORMANCE PREDICTABILITY

Tese do programa de doutoramento em Ciências e Tecnologias da Informação orientada por Professor Doutor Pedro Nuno San-Bento Furtado e apresentada ao Departamento de Engenharia Informática da Faculdade de Ciências e Tecnologia da Universidade de Coimbra

2014

· U C ·

Universidade de Coimbra

FCTUC FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

Massively Scalable Data Warehouses with Performance Predictability

by

João Pedro Matos da Costa

A dissertation submitted to the University of Coimbra
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in
Information Science and Technology

Adviser:

Professor Doutor Pedro Nuno San-Bento Furtado

Assistant Professor at University of Coimbra

Coimbra, Portugal
September, 2014

Financial support by:

Portuguese Foundation for Science and Technology (Fundação para a Ciência e a Tecnologia – co-financiamento do Programa Operacional Potencial Humano/POPH e da União Europeia, Programa de apoio à formação avançada de docentes do Ensino Superior Politécnico) through the PhD grant SFRH/BD/49892/2009.

“A satisfação está no esforço e não apenas na realização final.”

Mahatma Gandhi

Abstract

Data Warehouse (DW) systems are a fundamental tool for the decision-making process. Today these systems have to deal with increasingly large data volumes, which is typically stored as a star schema. The query workload is also more demanding, involving more complex, ad-hoc and unpredictable query patterns, with more simultaneous queries being submitted and executed concurrently.

Modern competitive markets require decisions to be taken in a timely fashion. It is not just a matter of delivering fast analysis, but also of guaranteeing that they will be available before business-decisions are made.

Moreover, the data volumes produced by data intensive industries are continuously increasing, stressing the processing infrastructure ability to provide such timely requirements even further. As a consequence, IT departments are continuously upgrading the processing infrastructure with the objective that, hopefully, the newer architecture will be able to deliver query results within the required time frame, but without any guarantees that it will be able to do so. There is no concrete method to define the minimal hardware requirements to deliver timely query results.

Several factors influence the ability of the DW infrastructure to provide timely results to queries, such as the query execution complexity (query selectivity, number of relations that have to be joined, the joins algorithms and the relations' size), the heterogeneity and capabilities of the processing infrastructure, including IO throughput, and the memory available to process joins and the implementation of the join algorithms. Larger data volumes and concurrent query loads; concurrent queries that are

executing simultaneously also influence the system ability to provide predictable execution times.

In spite of all the time and effort to come up with a parallel infrastructure to handle such increase in data volume and to improve query execution time, it may be insufficient to provide timely execution, particularly for ad-hoc queries. The performance of well-known queries can be tuned through a set of auxiliary strategies and mechanisms, such as materialized views and index tuning. However, for ad-hoc queries, such mechanisms are not an alternative solution. The query patterns unpredictability result in unpredictable query execution times, which may be incompatible with business requirements.

This dissertation proposes a data warehousing architecture that provides scalability and timely results for massive data volumes. The architecture is able to do this even in the presence of a large number of concurrent queries, and it is able to meet near real-time requirements. The ability to provide timely results is not just a performance issue (high throughput), but also a matter of returning query results when expected, according to the nature of the analysis and the business decisions.

Query execution is highly influenced by the number of relations that have to be joined together, the relations' size and the query selection predicates (selectivity), influencing the data volume that has to be read from storage and joined. This data volume and the memory available for joins, influence both the join order and the used join algorithms. These results in unpredictable execution times. The data volume is another factor of unpredictability, since there is no simple and accurate method to determine the impact of larger data volumes in query execution time.

To handle the unpredictability factors related to joining relations, we proposed the ONE data model, where the fact table and data from corresponding dimensions are physically stored into a single de-normalized relation, without primary and foreign keys, containing all the attributes from both fact and dimension tables. ONE trades-off storage space for a simpler and predictable processing model.

To provide horizontal scalability, we partitioned the de-normalized ONE relation into data fragments and distribute them among a set of processing nodes for parallel processing, yielding improved performance speedup. ONE delivers unlimited

data scalability, since the whole data (fact and dimensions), and not just the fact table, is linearly partitioned among nodes (with η nodes, each will have $1/\eta$ of the ONE node). Therefore, since the addition of more nodes to the processing infrastructure does not require additional data replication of dimensions, ONE provides massive data scalability.

By ensuring a linear distribution of the whole data, and not just the fact table, query execution time is improved proportionally to the data volume in each node. Moreover, since data in each node is already joined and thus query processing does not involve the execution of costly join algorithms, the speedup in each node is enhanced (almost) linearly as a function of the data volume that it has to process.

By de-normalizing the data, we also decrease the nodes' requirements, in what concerns physical memory (needed for processing joins), and query processing tasks, since the join processing tasks that were repeatedly (over and over) processed are removed. The remaining tasks, such as filtering and aggregations, have minimum memory and processing requirements. Only group by aggregations and sorting have memory requirements.

The concept of timely results (right-time execution) is introduced, and we propose mechanisms to provide right-time guarantees while meeting runtime predictability and freshness requirements.

The ability to provide right-time data analysis is gaining increasing importance, with more and more operational decisions being made using data analysis from the DW. The predictability of the query execution tasks is particularly relevant for providing right-time or real-time data analysis. We define right-time as the ability to deliver query results in a timely manner, before they are required. The aim is not to provide the fastest answers, but to guarantee that the answers will be available when expected and needed.

We proposed a Timely Execution with Elastic Parallel Architecture (TEEPA) which takes into consideration the query time targets to adjust and rebalance the processing infrastructure and thus providing right-time guarantees. When the current deployment is unable to deliver the time targets, it adds more processing nodes and redistributes the data volumes among them. TEEPA continuously monitors the local

query execution, the IO throughput and the data volume allocated to each processing node, to determine if the system is able to satisfy the user specified time targets.

TEEPA was designed to handle heterogeneous nodes and thus it takes into account their IO capabilities when performing the necessary data rebalancing tasks. The data volume allocated to each node is adjusted as a function of the whole data load (total number of tuples), the tuple size and the node' sequential scan throughput, with larger data volumes allocated to faster processing nodes. The node allocation (selection and integration of newer nodes) and data rebalancing tasks are continuously executed until the time targets can be assured.

There is an increasing demand for data analysis over near real-time data, with low latency and minimum freshness, which requires data to be loaded more frequently or loaded in a row-by-row fashion. However, traditionally DWs are periodically refreshed in batches, to reduce IO loading costs and costs related to the refreshing indexes and pre-computed aggregation data structures. Main memory DBMS eliminate IO costs and thus can handle higher data loading frequencies. However, physical memory is limited in size and cannot typically hold the whole tables and structures.

To provide freshness guarantees, the proposed architecture combines a parallel ONE deployment with an in-memory star schema model holding recent data. The in-memory part (O_s) maintains the recently loaded data, to allow the execution of real-time analyses. By using a star schema model in O_s, existing DW applications can be easily replaced and integrated with the architecture without the need to recreate the existing ETL tasks. Data is loaded into the in-memory O_s and remains there for real-time processing while there is memory available, so that the most recent data is held in the star schema. When the physical memory is exhausted, the data in O_s stored in the star schema model is moved to O_d in the ONE data model.

From the user perspective and data presentation, the architecture offers a logical star schema model view of the data, in order to provide easy integration with existing applications and because the model has advantages in what concerns users understanding and usability. A logical to physical layer manages data and processing between the O_s and O_d parts, including the necessary query rewriting for querying the data stored in each part, and merging of results.

Finally we present the mechanisms of the architecture that allow it to still guarantee right-time execution in the presence of huge concurrent query loads. Modern DWs also suffer from workload scalability limitations, with more and more queries (in particular ad-hoc) being concurrently submitted. Larger parallel infrastructures can reduce this limitation, but its scalability is constrained by the query-at-time execution model of custom RDBMs, where each query is individually processed, competing for resources (IO, CPU, memory,…) and accessing the common base data, without data and processing sharing considerations.

We propose SPIN, a data and processing sharing model that delivers predictable execution times for concurrent queries and overcomes the memory and scalability limitations of existing approaches. SPIN views the ONE relation in a node, as a logical circular relation, i.e. a relation that is constantly scanned in a circular fashion. When the end is reached, it continues scanning from the beginning, while there are queries running. Each query processes all the required tuples of relation ONE, but the scanning and the query processing does not start from the same first physical row. As the relation is read in a circular fashion, the first logical row is the one that already is cached in memory. The remaining tuples of the query are processed as they are being read from storage until the first logical row is reached. Data is read from storage and placed into an in–memory pipeline to be shared by all running concurrent queries.

IO reading cost is constant and is shared between running queries. Therefore, the submission of additional queries does not incur in additional IO costs and joins operations. The execution time of concurrent queries is influenced by the number and complexity of the query constraints (filtering) and the cost of aggregations.

To provide massive workload scalability, it shares data and processing among queries, by combining the running queries in logical query branches for filtering clauses and by extensive reuse of common computations and aggregation operations. It analyses the query predicates, and if there exists a logical branch in the current workload processing tree with common predicates, it is registered in that logical branch, and the corresponding query predicates are removed. Otherwise, if it does not exist a logical branch that meets the query predicates, it is registered as a new logical branch of the base data pipeline. This enhances processing sharing, and reduces the number of filtering conditions. The architecture has a branch optimizer that is continuously

adjusting the number and order of the existing branches, and reorganizing them as required. Whenever possible, a query can merge and combine the results that are being processed by other branches, thus simplifying and reducing the data volume that the query branch has to filter and to process.

Since tuples flow using the same reading order, if data doesn't change, the evaluation of the branch predicates against every tuple that flows along the branch will not change. The result of predicate evaluation will be the same as the last time it was evaluated. To avoid subsequent evaluation of unchanged data tuples, we extended the SPIN approach with a *bitset* processing approach. A branch *bitset* (bitmap) is built according to the branch' predicates, where each bit represents the boolean result of the predicate evaluation (true/false) applied to a corresponding tuple index. Future evaluations of the tuple can take advantage of the existence of this *bitset*, since the selection operator that evaluates the predicate can be replaced by a fast lookup operator to the corresponding position in the *bitset* to gather the result. *Bitsets* are small and reside in memory in order to avoid introducing overhead at IO level. This is particularly relevant for predicates with high evaluation costs.

Through the analysis of the data path (branches) of queries, and the required computational costs of each branch, it is possible to determine high accurate estimations of query execution times. Therefore, predictable execution times can be given for massive workload scalability.

Tighter right-time guarantees can be provided by extending the parallel infrastructure, and redistributing data among processing nodes, but also by redistributing queries, query processing and data branches between nodes holding replicated fragments. This is achieved by using two distinct approaches, a parallel fine-tuned fragment level processing, named CARROUSEL, and an early-end query processing mechanism.

CARROUSEL is a flexible fragment processor that uses idle nodes, or nodes currently running less time-stricter queries, to process some of the fragments required by time-strict queries, on behalf of the fragment node's owner. By reducing the data volume to be processed by a node, it can provide faster execution times. Alternatively, it may distribute some logical data branches among nodes with replicated fragments,

and thus reduce query processing. This is only possible with nodes having replicated data fragments.

The execution of a query ends when all tuples of the data fragments are processed and the circular logical loop is completed. But as the system is continuously spinning, reading and processing over and over the same data, it collects insightful information regarding the data that is stored in each data fragment. For some logical data branches, this can be relevant to reduce memory and computational usage by using a postponed start (delaying the query execution until the first relevant fragment is loaded) and early-end approaches (detaching the query pipeline when all the relevant fragments for a query have been processed). This information is useful when the architecture needs to perform a data rebalancing process, with the rebalanced data being clustered according to logical branch predicates and stored as new data fragments.

Resumo Alargado

Data Warehouses (DW) são ferramentas fundamentais no apoio ao processo de tomada de decisão, que processam volumes de dados cada vez maiores, e que normalmente são armazenados usando um modelo em estrela (*star schema*). Os resultados das pesquisas e das análises devem estar disponíveis em tempo útil. No entanto, a complexidade das pesquisas que são submetidas é cada vez mais exigente, com padrões de pesquisa imprevisíveis (*ad-hoc*) e a execução simultânea de pesquisas faz com que o tempo de execução seja imprevisível. A Fig. 1 ilustra os fatores de imprevisibilidade que influenciam a capacidade de fornecer resultados em tempo útil.

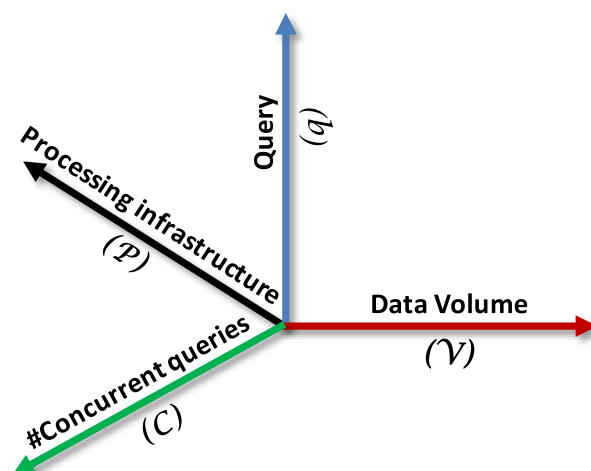


Fig. 1 – Fatores que influenciam a capacidade de fornecer resultados em tempo útil

Mercados competitivos requerem que os resultados sejam disponibilizados em tempo útil, para ajudar o processo de tomada de decisão. Não é apenas uma questão de rapidez na obtenção dos resultados, mas de garantir que estes ficam atempadamente

disponíveis para a tomada de decisão. Estratégias de pré-computação das pesquisas podem ajudar na obtenção de resultados mais rápidos, no entanto a sua utilização é limitada apenas a pesquisas com padrões conhecidos. As consultas com padrões de pesquisa imprevisíveis (*ad-hoc*) são executadas sem quaisquer garantias de tempo de execução.

Existem vários fatores que influenciam a capacidade da DW fornecer resultados às pesquisas em tempo útil, como a seletividade da pesquisa, número de tabelas que necessitam de ser relacionadas, os algoritmos de junção e o tamanho das tabelas, a heterogeneidade e a capacidade da infraestrutura de processamento, incluindo a velocidade de leitura de disco, e a memória disponível para a junção das tabelas. O aumento do volume de dados e do número de pesquisas que estão em execução simultânea, também influenciam a capacidade do sistema em fornecer tempos de execução previsíveis.

Foram propostas diversas infraestruturas de processamento paralelo com capacidade para lidar com o aumento do volume de dados, e melhorar o tempo de execução das pesquisas, no entanto estas não permitem garantir a disponibilização atempada dos resultados, particularmente das pesquisas *ad-hoc*. O tempo de execução de pesquisas com padrões conhecidos pode ser otimizado através de um conjunto de estratégias e mecanismos auxiliares, tais como, a utilização de vistas de materializadas e índices. No entanto, para consultas *ad-hoc*, tais mecanismos não são uma solução. A imprevisibilidade do padrão de pesquisas origina tempos de execução imprevisíveis, que podem ser incompatíveis com os requisitos de negócio.

Em muitos negócios, o crescente volume de dados condiciona ainda mais a capacidade da infraestrutura de processamento de fornecer resultados em tempo útil. Como consequência, os departamentos de TI têm a necessidade de atualizar, com frequência, a infraestrutura de processamento, na expectativa que esta consiga processar atempadamente as pesquisas, mas sem garantia de que o consiga. Não existe um método concreto que permita definir os requisitos mínimos de *hardware* que permita a execução atempada das pesquisas.

Esta dissertação propõe uma arquitetura de *Data Warehouse* escalável com capacidade de lidar com grandes volumes de dados e de fornecer resultados em tempo útil, mesmo quando um grande número de pesquisas estão a ser simultaneamente

executadas. A capacidade de fornecer resultados em tempo útil não é apenas uma questão de desempenho, mas também uma questão de ser capaz de retornar atempadamente os resultados às pesquisas, quando esperado, de acordo com a natureza da análise e as decisões de negócios.

A complexidade da execução de uma pesquisa é influenciada por vários fatores, tais como a seletividade da pesquisa, o tamanho das tabelas, o número de junções e os algoritmos de junção. O volume de dados e memória disponível influenciam tanto a ordem de junção como o algoritmo de junção utilizado, resultando em custos de execução imprevisíveis. A necessidade de juntar as tabelas de dimensão com a tabela de factos advém do modelo em estrela (*star schema*). O volume de dados é outro fator de imprevisibilidade, não sendo possível determinar com precisão o impacto do aumento do volume de dados no tempo de execução das pesquisas.

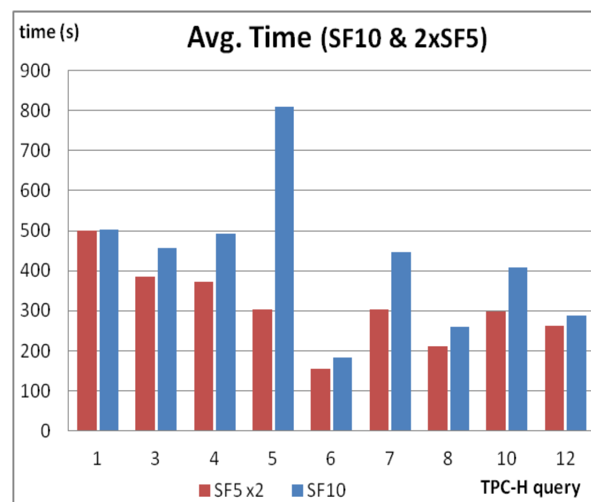


Fig. 2 – Variação do tempo de execução com o aumento (2x) do volume de dados

A Fig. 2, extraída da secção experimental, mostra a estimativa do tempo de execução (a vermelho) e o tempo real (a azul) para a execução de um conjunto de pesquisas da *benchmark* do TPC-H, quando o volume de dados (neste caso 5GB, fator de escala SF=5) aumenta para o dobro (10GB, fator de escala SF=10).

Para lidar com os fatores de imprevisibilidade relacionados com a junção de tabelas, propusemos o modelo de dados desnormalizado, chamado ONE. Neste modelo, os dados da tabela de factos, assim como os correspondentes dados das tabelas de dimensão, são fisicamente guardados numa única tabela desnormalizada, como ilustrado na Fig. 3, contendo todos os atributos das tabelas, exceto as chaves

estrangeiras da tabela de factos e chaves artificiais das tabelas de dimensão. O modelo de dados ONE requer mais espaço para guardar os dados, no entanto o modelo de processamento é mais simples e oferece tempos de execução previsíveis.

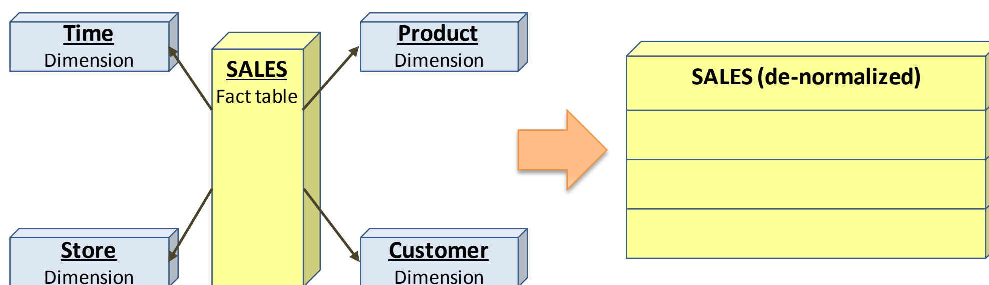


Fig. 3 – O modelo em estrela vs o modelo ONE

No modelo de dados ONE, a tabela desnormalizada pode ser particionada em fragmentos de dados mais pequenos e distribuídos, sem *overheads,* pelos nós da infraestrutura de processamento paralelo. ONE possibilita uma escalabilidade quase ilimitada, uma vez que a totalidade dos dados, dos factos e das dimensões (e não apenas da tabela de factos), é linearmente dividida pelos nós da infraestrutura de processamento (com η nós homogéneos, cada nó terá 1/η dos dados). A Fig. 4 ilustra o particionamento e distribuição dos fragmentos do modelo de dados ONE numa infraestrutura de processamento paralelo.

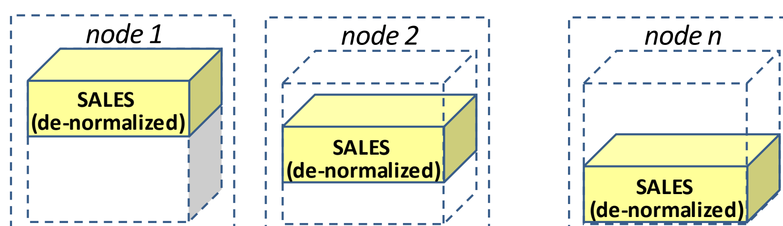


Fig. 4 – Particionamento e distribuição dos fragmentos do modelo ONE pelos nós

Ao garantir uma distribuição linear de todos os dados, e não apenas os dados da tabela de factos, o tempo de execução das pesquisas é melhorado proporcionalmente ao volume de dados existente em cada nó. Além disso, e porque os dados estão desnormalizados, o processamento das pesquisas é bastante simplificado e previsível, pois fica reduzido às operações de filtragem e de agregação dos dados, o que reduz os requisitos da infraestrutura de processamento.

Por norma, quando uma pesquisa é submetida, não existe uma noção clara de quanto tempo irá demorar e se o resultado será obtido em tempo útil, por exemplo antes da tomada de decisão. Definimos o conceito de execução em tempo útil (*right-time*) como a capacidade de executar pesquisas de modo que os resultados estejam

disponíveis antes de um determinado objetivo temporal (execução atempada). O objetivo não é obter execuções mais rápidas, mas sim garantir que os resultados estarão disponíveis quando esperado. São propostos mecanismos que permitem fornecer previsibilidade de tempo de execução de pesquisas e garantias de execução atempada das que tenham objetivos temporais.

Para pesquisas que tenham objetivos temporais inferiores ao oferecido pela atual infraestrutura de processamento, propusemos um modelo de processamento chamado TEEPA (*Timely Execution with Elastic Parallel Architecture*), que toma em consideração os objetivos temporais das pesquisas para ajustar e rebalancear a infraestrutura de processamento até que estes sejam atingidos. Quando a infraestrutura atual não consegue executar atempadamente as pesquisas, são adicionados novos nós de processamento e o volume de dados é posteriormente redistribuído entre eles. TEEPA monitora continuamente a execução da pesquisa, o volume de dados alocado em cada nó, e a taxa de transferência IO, para determinar se as pesquisas são executadas atempadamente.

Para nós de processamento heterogéneos, TEEPA toma em consideração as capacidades de IO de cada nó para determinar quantos nós adicionais são necessários e qual a redistribuição de dados. O volume de dados alocado em cada nó é ajustado em função do volume total (número total de registos), do tamanho do registo e da taxa de transferência de cada nó. Deste modo, a nós mais rápidos são atribuídos maiores volumes de dados. O processo de seleção e integração de novos nós e posterior reequilíbrio dos dados é executado até que os objetivos temporais sejam atingidos.

Por outro lado, a necessidade de analisar dados obtidos quase em tempo real, com mínima latência e frescura (*freshness*), é cada vez maior, o que requer que os dados sejam carregados mais frequentemente, à medida que são registados. Tipicamente as DW são refrescadas em *batch*, de modo a reduzir os custos de carregamento e os custos relacionados com o refrescamento de estruturas relacionadas, como índices e vistas materializadas. Sistemas de base de dados em memória minimizam estes custos, e possibilitam que os dados sejam carregados mais frequentemente. No entanto, a memória é finita e insuficiente para conter a totalidade dos dados.

De modo a oferecer latência mínima, os dados são divididos em duas partes distintas: os dados antigos são guardados num modelo de dados ONE, ao qual

chamámos O_d, e os dados mais recentes são guardados em memória num modelo, designado de O_s. Os dados podem ser carregados com maior frequência para O_s, reduzindo a sua latência, e são mantidos aí enquanto existir memória disponível. Quando necessário, por exemplo para libertar memória e permitir a inserção de novos dados, os dados mais antigos existentes em O_s são movidos para O_d, como ilustrado na Fig. 5.

A existência deste modelo híbrido, composto por O_d e O_s, permite que as DW existentes, que utilizam o modelo em estrela, possam ser substituídas por este modelo com mínimo impacto ao nível dos processos de ETL.

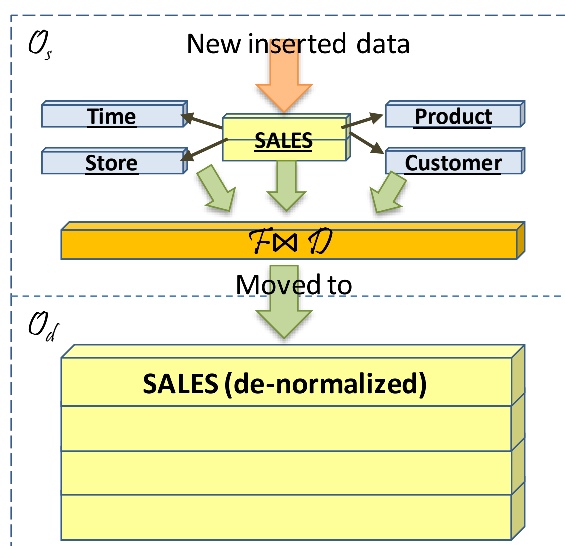


Fig. 5 – Fluxo dos dados entre O_d e O_s

Na perspetiva do utilizador e das aplicações, este modelo híbrido oferece uma visão lógica dos dados num modelo em estrela, possibilitando uma fácil integração com aplicações e processos de carregamento existentes, e oferecendo as vantagens do modelo em estrela, nomeadamente ao nível de usabilidade e facilidade de utilização. Uma camada de abstração gere a coerência de dados e processamento entre as duas componentes (O_s e O_d), incluindo a reescrita das pesquisas, de modo a processar os dados que se encontram em cada uma das componentes.

Nesta tese são também propostos mecanismos que oferecem garantias de execução atempada de pesquisas, mesmo quando um grande número de pesquisas está a ser processado simultaneamente. Infraestruturas paralelas podem minimizar esta questão, no entanto a sua escalabilidade é constrangida pelo modelo de execução dos sistemas de bases de dados relacionais, onde cada pesquisa é processada isoladamente e

competindo com as outras pelos recursos (IO, CPU, memória, …). É proposto um modelo de processamento de pesquisas, chamado SPIN, que analisa as pesquisas submetidas e, sempre que possível, partilha dados e processamento entre elas, conseguindo assim oferecer tempos de execução previsíveis. SPIN utiliza o modelo de dados ONE, mas considera a tabela como sendo circular (tabela lida continuamente de uma forma circular). Enquanto existirem pesquisas a serem executadas, os dados são lidos sequencialmente, e quando chega ao fim da tabela, recomeça a lê-los desde o início. À medida que os dados são lidos, são colocados sequencialmente numa janela deslizante em memória (*base pipeline*), para serem partilhados pelas várias pesquisas, como ilustrado na Fig. 6. Cada pesquisa processa todos os registos da tabela, no entanto a leitura e o processamento não começa no registo número 1, mas sim no primeiro registo da janela deslizante (início lógico). Os restantes registos são processados à medida que forem lidos e colocados na janela deslizante, até que o próximo registo a ser processado seja o do início lógico, isto é, após um ciclo completo.

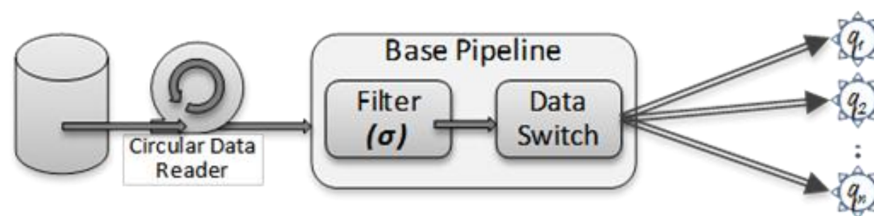


Fig. 6 – Modelo de leitura e partilha de dados pelas pesquisas

O custo da leitura dos dados é constante e partilhado por todas as pesquisas. Deste modo, a submissão de novas pesquisas não introduz custos adicionais ao nível da leitura de dados. O tempo de execução das pesquisas é influenciado pela complexidade e número dos filtros (restrições) das pesquisas e pelo custo das agregações e ordenações dos dados. SPIN partilha dados e processamento entre pesquisas, combinando filtros e computações comuns a várias pesquisas num único fluxo (ramo) de processamento. Os vários ramos (branches) são sequencialmente conectados, formando uma estrutura em árvore que denominámos de WPtree (Workload Processing Tree), tendo como raiz o base pipeline. Quando uma pesquisa é submetida, se existir um ramo de processamento com predicados comuns aos da pesquisa, a pesquisa é encadeada como um novo ramo desse ramo comum, e são removidos os respetivos predicados da pesquisa. Se não existir um ramo com predicados comuns, a pesquisa é encadeada como um novo ramo do base pipeline. Deste modo, reduz-se o volume de dados que está em memória para

processamento, bem como o custo de processamento dos predicados. A Fig. 7 ilustra um exemplo de uma árvore de processamento.

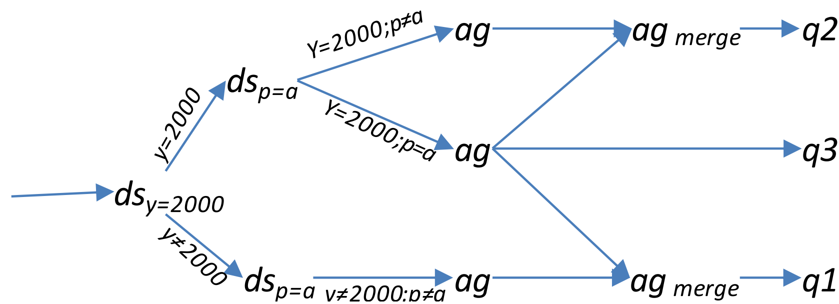


Fig. 7 – Exemplo de uma árvore de processamento

A árvore de processamento é continuamente monitorizada, e quando necessário, um optimizador reorganiza dinamicamente o número e a ordem dos ramos. Sempre que possível, uma pesquisa é processada através da combinação dos resultados que estão a ser processados por outros ramos, deste modo simplificando e reduzindo o volume de dados que a pesquisa tem que processar.

Como os registos são lidos e processados pela mesma ordem, enquanto os dados não forem alterados, o resultado da avaliação dos predicados de cada registo é o mesmo que o da última vez que foi avaliado. Para evitar o custo da avaliação de registos anteriormente avaliados, e que não foram alterados, é proposta uma extensão ao modelo de processamento SPIN que utiliza uma abordagem de processamento baseada em *bitsets* (estruturas similares aos índices *bitmaps*). Para cada ramo é criado um *bitset* com o resultado da avaliação dos seus predicados, sendo o resultado de cada registo guardado na correspondente posição do *bitset*. Após o bitset estar completo, a posterior avaliação desses predicados pode ser substituída por uma simples verificação no *bitset*. Os *bitsets* têm um tamanho reduzido e são guardados em memória para evitar a introdução de custos adicionais ao nível de IO. *Bitsets* são particularmente relevantes para predicados complexos e com elevado custo de processamento, sendo criados e removidos dinamicamente de acordo com uma política de retenção, que toma em consideração vários aspetos, tais como a memória disponível, cardinalidade, e o custo da avaliação dos predicados.

Através da análise do conjunto sequencial de ramos de uma pesquisa (*path*), e dos custos de processamento de cada ramo, é possível estimar, com elevada precisão, o tempo de execução da pesquisa, mesmo quando existe um grande número de pesquisas

a serem executadas simultaneamente. Para satisfazer pesquisas com objetivos temporais mais exigentes, é proposto um mecanismo de processamento, denominado CARROUSEL, que além de redistribuir e/ou replicar fragmentos dos dados pelos vários nós de processamento, redistribui também o processamento das pesquisas e dos ramos pelos nós. Tomando em consideração os *bitsets* existentes, é possível determinar quais os fragmentos de dados que cada pesquisa necessita processar e deste modo reduzir o custo de processamento através da ativação/desativação dinâmica dos ramos (consoante os fragmentos que estão nesse momento em memória). É possível terminar antecipadamente a execução de uma pesquisa, antes do término do ciclo.

CARROUSEL é um processador flexível de fragmentos que utiliza nós inativos, ou nós que estão a executar pesquisas com objetivos temporais menos exigentes, para processar em paralelo alguns dos fragmentos de dados requeridos por pesquisas com objetivos temporais mais exigentes, como ilustrado na Fig. 8. Ao reduzir-se o volume de dados que é processado por cada nó, consegue-se tempos de execução mais rápidos. Alternativamente, alguns dos ramos de processamentos podem ser redistribuídos por nós com fragmentos replicados.

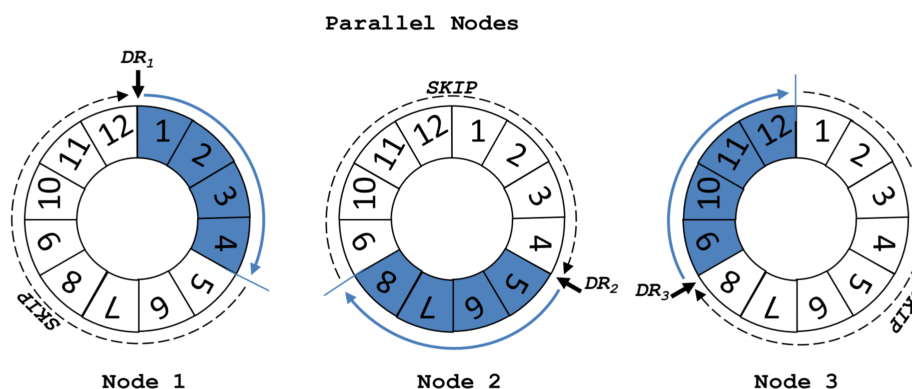


Fig. 8 – CARROUSEL - processamento de fragmentos

A execução da pesquisa termina quando todos os registos foram processados. No entanto, como os dados estão continuamente a ser lidos e à medida que são processados é recolhida informação relevante sobre os dados que existem em cada fragmento. Esta informação é relevante para decidir quais os fragmentos e os ramos a processar que devem ser redistribuídos pelos nós, por forma a reduzir custos de processamento e tempos de execução.

Acknowledgement

First, I would like to thank to Professor Pedro Furtado for his advice. During the last years, he continuously encouraged me and was always ready to discuss directions and objectives. His critical comments and observations helped me to keep moving forward with this work.

I gratefully acknowledge the funding sources that made my Ph.D. work possible. I was partially funded by the Portuguese Foundation for Science and Technology (SFRH/BD/49892/2009) and the by IPC-ISEC (Polytechnic Institute of Coimbra - Coimbra Institute of Engineering).

Finally, I would like to dedicate this thesis to my wife Renata, who always kept motivating and supporting me, and to my sons Diogo and Luis. I thank them for their love and patient, which were indispensable.

Thank you all.

Publications

"Data Warehouse Processing Scale-up for Massive Concurrent Queries with SPIN", João Pedro Costa & Pedro Furtado, in TLDKS Journal, Transactions on Large-Scale Data- and Knowledge-Centered Systems, Springer 2015, ISBN 978-3-662-46334-5.

"Improving the Processing of DW Star-Queries under Concurrent Query Workloads", João Pedro Costa & Pedro Furtado, In Proceedings of the 16th International Conference on Data Warehousing and Knowledge Discovery - DaWaK 2014, Munich, Germany. Springer 2014, ISBN 978-3-319-10159-0

"SPIN: Concurrent Workload Scaling over Data Warehouses", João Pedro Costa & Pedro Furtado, In Proceedings of the 15th International Conference on Data Warehousing and Knowledge Discovery, DaWaK 2013. Prague, Czech Republic. Springer 2013, ISBN 978-3-642-40130-5

"Providing Timely Results with an Elastic Parallel DW", João Pedro Costa, Pedro Martins, José Cecilio & Pedro Furtado, In Proceedings of the 20th International Conference on Foundations of Intelligent Systems, ISMIS'12, Macau, China. Springer, ISBN 978-3-642-34623-1

"TEEPA: a Timely-aware Elastic Parallel Architecture", João Pedro Costa, Pedro Martins, José Cecilio & Pedro Furtado, In Proceedings of the 16th International Database Engineering & Applications Symposium, IDEAS'12, Prague, Czech Republic, 2012. ACM 2012, ISBN 978-1-4503-1234-9.

"Overcoming the Scalability Limitations of Parallel Star Schema Data Warehouses", João Pedro Costa, José Cecílio, Pedro Martins & Pedro Furtado, In Proceedings of the 12th International Conference on Algorithms and Architectures for Parallel Processing, ICA3PP'12, Fukuoka, Japan. Springer, ISBN 978-3-642-33077-3

"A Predictable Storage Model for Scalable Parallel DW", João Pedro Costa, Pedro Martins, José Cecílio, & Pedro Furtado, In Fifteenth International Database

Engineering and Applications Symposium (IDEAS 2011), Lisbon, Portugal. ACM 2011, ISBN 978-1-4503-0627-0

"ONE: a Predictable and Scalable DW Model", João Pedro Costa, José Cecílio, Pedro Martins & Pedro Furtado, In Proceedings of the 13th International Conference on Data Warehousing and Knowledge Discovery, DaWaK'11, Toulouse, France. Springer 2011, ISBN 978-3-642-23543-6

"Blending OLAP Processing with Real-Time Data Streams", João Pedro Costa, José Cecílio, Pedro Martins & Pedro Furtado, In the 16th International Conference Database Systems for Advanced Applications, DASFAA 2011, Hong Kong, China, April 22-25, 2011. Springer 2011, ISBN 978-3-642-20151-6

"StreamNetFlux: Birth of Transparent Integrated CEP-DBs", João Pedro Costa, Pedro Martins, José Cecílio & Pedro Furtado, in proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems, DEBS 2010, Cambridge, United Kingdom, July 12-15, 2010. ACM 2010, ISBN 978-1-60558-927-5

"Towards a QoS-aware DBMS", João Pedro Costa & Pedro Furtado, in proceedings of the IEEE 24th International Conference on Data Engineering Workshops, the 3rd International Workshop on Self-Managing Database Systems (SMDB), Cancun, México. IEEE Computer Society 2008

Table of Contents

Abstract ---------- i
Resumo Alargado ---------- ix
Acknowledgement ---------- xix
Publications ---------- xxi
Table of Contents ---------- xxiii
List of Figures ---------- xxvii
List of Tables ---------- xxxi
List of Acronyms ---------- xxxiii

1 Introduction ---------- 1
1.1 Problem Statement and Research Goals ---------- 2
1.2 Dissertation Contributions ---------- 5
1.3 Organization of the Dissertation ---------- 7

2 Background and Related Work ---------- 9
2.1 Data Warehouse Background ---------- 9
2.1.1 Data Warehouse models ---------- 10
2.1.2 Typical aggregation queries ---------- 11
2.2 Query processing ---------- 12
2.2.1 Join processing ---------- 13
2.2.2 Mechanisms for improving query performance ---------- 14
2.3 De-normalization and decomposition ---------- 15
2.4 Data sharing and Pipeline Processing ---------- 17
2.5 Parallel DW, Data Partitioning and Placement ---------- 19
2.6 Predictable execution time ---------- 24

3 A Timely and Massively Scalable DW ---------- 27
3.1 Providing predictable execution ---------- 28
3.2 Mechanisms for Providing Unlimited Scalability ---------- 29
3.3 Mechanisms for Providing Right-time ---------- 32
3.4 Providing Freshness guarantees ---------- 34
3.5 Handling massive concurrent queries (query workload) ---------- 35
3.6 Performance and auxiliary data structures ---------- 39
3.7 Limitations of the proposed mechanisms ---------- 39
3.8 Chapter Summary ---------- 40

4 A Scalable and Predictable Data Model ---------- 41
4.1 Unpredictability factors ---------- 42
4.2 The ONE Data Model ---------- 43
4.3 Handling scalable data volumes ---------- 46
4.4 Providing Freshness ---------- 46

4.5 ONE Query Processing ---------- **48**
4.6 Changes to the ETL process ---------- **50**
4.7 Storage size requirements ---------- **51**
4.8 Partial de-normalization ---------- **53**
4.8.1 Partial de-normalization of larger dimension tables ---------- 53
4.8.2 Workload-driven de-normalization ---------- 56
4.9 Chapter Summary ---------- **58**

5 Providing Right-time with a Elastic Parallel Architecture ---------- **59**
5.1 Introduction ---------- **60**
5.2 Speeding up the ONE data model – ONE-P ---------- **61**
5.3 TEEPA - Timely Execution with an Elastic Parallel Architecture ---------- **64**
5.3.1 TEEPA Manager ---------- 68
5.3.2 Node Processing Service (TEEPA - NS) ---------- 71
5.4 Logical to Physical Translator ---------- **72**
5.5 Data Allocation to Heterogeneous Nodes ---------- **74**
5.6 Detection of node unbalancing and overloading ---------- **75**
5.6.1 Resolving overloads by rebalancing data among nodes ---------- 76
5.6.2 Resolving overload by allocation of additional nodes ---------- 77
5.7 Chapter Summary ---------- **79**

6 SPIN: Concurrent Workload Scaling over Data Warehouses ---------- **81**
6.1 SPIN Processing Model ---------- **82**
6.2 SPIN query handling ---------- **84**
6.3 SPIN operators and data processing pipelines ---------- **85**
6.4 Query processing path ---------- **87**
6.5 Building the Workload Processing Tree ---------- **89**
6.6 Reorganization of the workload processing tree ---------- **93**
6.7 Handling data updates and deletes ---------- **95**
6.8 SPIN Prototype ---------- **96**
6.8.1 The data access layer ---------- 97
6.8.2 The query handler layer ---------- 98
6.8.3 The SPIN processing layer ---------- 99
6.9 Chapter Summary ---------- **99**

7 Providing Right-Time Guarantees to Scalable Concurrent Workloads ---------- **101**
7.1 SPIN predicate evaluation overload ---------- **101**
7.2 The bitset branch processing approach ---------- **103**
7.2.1 Creation of Bitsets ---------- 105
7.2.2 Bitset lookup operators ---------- 106
7.2.3 Mixed branch processing: branches with and without bitsets ---------- 107
7.2.4 Merging bitsets along the query logical path ---------- 108
7.2.5 Pushing forward bitsets to the data reader ---------- 109
7.2.6 In-Memory Bitmap Management and Retention ---------- 110
7.3 Optimizing the processing of data fragments ---------- **112**
7.4 Parallel Processing - CARROUSEL ---------- **116**
7.4.1 Managing fragment metadata ---------- 117
7.4.2 Balancing Data and Query processing among nodes ---------- 119

7.4.3 Rebalancing processing data load ---------- 120
7.4.4 Rebalancing processing load ---------- 122
7.5 Fragment level data reorganization ---------- 123
7.6 Chapter Summary ---------- 125

8 Experimental Evaluation ---------- 127
8.1 Experimental Setup and benchmarks ---------- 128
8.1.1 Benchmark ---------- 128
8.1.2 Data Schemas ---------- 129
8.1.3 Query workload ---------- 130
8.1.4 Processing Infrastructure ---------- 131
8.1.5 DBMS engines ---------- 131
8.2 Storage requirements of the ONE data model ---------- 132
8.2.1 Storage overheads of the star schema model ---------- 132
8.2.2 Storage size in each node of a parallel infrastructure ---------- 133
8.3 Evaluation of Execution Time of ONE ---------- 135
8.3.1 Predictable performance with scalable data volumes ---------- 135
8.3.2 Execution Time (single node with Oracle) ---------- 138
8.3.3 Execution time variability ---------- 140
8.4 Evaluation of ONE-P ---------- 141
8.4.1 Execution time (in each node) ---------- 141
8.4.2 Speedup with larger processing infrastructures ---------- 143
8.4.3 Impact of query selectivity in performance ---------- 144
8.4.4 Inter-node variance of query execution time ---------- 144
8.4.5 Cost of exchanging partial results ---------- 145
8.5 TEEPA Right-Time Evaluation ---------- 147
8.6 SPIN Evaluation ---------- 150
8.6.1 Influence of number of queries in query performance ---------- 150
8.6.2 Influence of number of queries in Throughput ---------- 152
8.6.3 Influence of the data volume in throughput ---------- 152
8.6.4 Influence of the workload query pattern in query performance ---------- 154
8.6.5 Evaluation of SPIN with bitset processing ---------- 155
8.7 Chapter Summary ---------- 158

9 Conclusions and Future Work ---------- 159
9.1 Conclusions ---------- 159
9.2 Future Work ---------- 161

References ---------- 163
Appendix A – TPC-H Queries ---------- 173

List of Figures

Fig. 1 – Fatores que influenciam a capacidade de fornecer resultados em tempo útil ---------------------------- ix
Fig. 4 – Particionamento e distribuição dos fragmentos do modelo ONE pelos nós ---------------------------- xii
Fig. 5 – Fluxo dos dados entre O_d e O_s ---------------------------- xiv
Fig. 6 – Modelo de leitura e partilha de dados pelas pesquisas ---------------------------- xv
Fig. 7 – Exemplo de uma árvore de processamento ---------------------------- xvi
Fig. 8 – CARROUSEL - processamento de fragmentos ---------------------------- xvii
Fig. 2.1 – The Data Warehouse environment ---------------------------- 10
Fig. 2.2 – An example of a sales star schema model ---------------------------- 11
Fig. 2.3 – Template of typical aggregation query ---------------------------- 12
Fig. 2.4 – Example execution plan ---------------------------- 12
Fig. 2.5 – Parallel approaches a) shared-memory b) shared-disk c) shared-nothing ---------------------------- 20
Fig. 2.6 – A parallel deployment of a star schema with replicated dimensions ---------------------------- 20
Fig. 2.7 – Query processing in shared-nothing ---------------------------- 21
Fig. 3.1 – Unpredictability factors that influence the ability to provide timely results ---------------------------- 27
Fig. 3.3 – Execution time for a 2x increase in data volume ---------------------------- 30
Fig. 3.4 – ONE fragments distributed among nodes ---------------------------- 30
Fig. 3.5 – TEEPA framework ---------------------------- 33
Fig. 3.7 – In-memory data buffering and ONE data flushing ---------------------------- 35
Fig. 3.8 – SPIN Data processing model ---------------------------- 36
Fig. 3.10 – Branch processing ---------------------------- 37
Fig. 3.11 – A high level integration view of the proposed mechanisms ---------------------------- 40
Fig. 4.1 – A typical star schema and an example of query execution plan ---------------------------- 42
Fig. 4.2 – Execution time of different TPC-H queries ---------------------------- 43
Fig. 4.4 – Query execution plan a) star schema b) ONE ---------------------------- 45
Fig. 4.5 – Execution time of different TPC-H queries with the ONE data model ---------------------------- 45
Fig. 4.6 – ONE fragments distribution among nodes ---------------------------- 46
Fig. 4.7 – The hybrid O_s and O_d data model ---------------------------- 47
Fig. 4.9 – Decomposition of query q ---------------------------- 49
Fig. 4.10 – Data size distribution of the TPC-H schema ---------------------------- 52
Fig. 4.11 – Partial de-normalized schema ---------------------------- 54
Fig. 4.12 – Example of a set of $\mathcal{A}_p$ with k=4 in a 2-dimensional data space ---------------------------- 57
Fig. 5.1 – ONE fragments distribution among nodes ---------------------------- 62
Fig. 5.2 – Speedup comparison between TPCH-P and ONE-P ---------------------------- 62

Fig. 5.3 – Data allocation size: a) evenly partitioned b) by nodes' performance ---------------------------------64
Fig. 5.4 – The timely execution triangle ---------------------------------64
Fig. 5.5 – TEEPA over an elastic set of heterogeneous nodes ---------------------------------65
Fig. 5.6 – TEEPA framework ---------------------------------66
Fig. 5.8 – TEEPA modules in detail ---------------------------------68
Fig. 5.9 – Syntax of the time target clause added to a SELECT statement ---------------------------------69
Fig. 5.10 – Syntax of the time target at session level ---------------------------------69
Fig. 5.11 – Additional node added ---------------------------------71
Fig. 5.12 – Storage space scope ---------------------------------73
Fig. 5.13 – a) ONE-P partitioning and placement b) ONE-P Query processing ---------------------------------74
Fig. 5.14 – Rebalancing the data volume when a new node is added---------------------------------76
Fig. 6.1 – SPIN base data processing model ---------------------------------82
Fig. 6.2 – SPIN Fragment Metadata---------------------------------83
Fig. 6.3 – SPIN sequential data reading loop ---------------------------------84
Fig. 6.4 – SPIN Data pipeline with operators ---------------------------------86
Fig. 6.5 – SPIN Logical Branch Processing Model ---------------------------------88
Fig. 6.6 – SPIN deployment of query-specific pipelines ---------------------------------89
Fig. 6.7 – Aggregation Branch processing ---------------------------------92
Fig. 6.8 – Number of times tuples are evaluated ---------------------------------93
Fig. 6.9 – WPTree reorganization with group removal (1)---------------------------------93
Fig. 6.10 – WPTree reorganization with group removal (2) ---------------------------------94
Fig. 6.11 – WPTree reorganization with group removal (3) ---------------------------------94
Fig. 6.12 – SPIN - handling deletes ---------------------------------95
Fig. 6.13 – SPIN - Handling updates ---------------------------------96
Fig. 6.14 – SPIN prototype diagram - release 1.6.3 (June2013) ---------------------------------97
Fig. 7.1 – Increasing processing costs for large WPtree --------------------------------- 102
Fig. 7.2 – An example of a *WPtree* --------------------------------- 104
Fig. 7.3 – Online predicate bitmap indexes--------------------------------- 104
Fig. 7.4 – Predicate evaluation and building the bitset--------------------------------- 105
Fig. 7.5 – A branch processing layout using bitsets --------------------------------- 106
Fig. 7.6 – Selection operators a) NOT b) NOR c) NOR--------------------------------- 107
Fig. 7.7 – Branch processing with a bitwise AND of the logical path branches --------------------------------- 108
Fig. 7.8 – Data Reader Bitset computed as a bitwise OR of the branches bitsets--------------------------------- 109
Fig. 7.9 – Early-end execution and query lifetime --------------------------------- 113
Fig. 7.10 – Data reader fragment skipping--------------------------------- 113
Fig. 7.11 – Carrousel (parallel processing) --------------------------------- 116
Fig. 7.12 – Data readers at different positions a) single node b) multiple nodes --------------------------------- 117
Fig. 7.13 – A parallel deployment with inactive fragments in each nodes that can be skipped------------ 119

Fig. 7.14 – Distribute Qr fragments and run WPtree in parallel ---------- 121
Fig. 7.15 – Replicate a subset of q fragments among nodes and run q in parallel ---------- 122
Fig. 7.16 – Split WPtree branches among nodes ---------- 123
Fig. 8.1 – TPC-H benchmark schema ---------- 128
Fig. 8.2 – TPC-H storage size distribution ---------- 133
Fig. 8.3 – Storage size distribution with different numbers of processing nodes (TPCH-P)) ---------- 134
Fig. 8.4 – Storage scalability of the schemas ---------- 135
Fig. 8.5 – TPCH execution times for large data volumes ---------- 136
Fig. 8.6 – ONE predictable execution time for larger data volumes ---------- 137
Fig. 8.7 – Error estimation in predicting the query execution time for higher data volumes ---------- 138
Fig. 8.8 – Average execution time for queries 1 .. 10 ---------- 138
Fig. 8.9 – Average execution time of ONE for varying SF for queries 1-10 (Oracle) ---------- 139
Fig. 8.10 – Execution time variability for varying SF ---------- 140
Fig. 8.11 – Data volume a) in each node b) total (sum each node) ---------- 141
Fig. 8.12 – Average time to process partial results (t_n) in each node ---------- 142
Fig. 8.13 – Execution time to compute the partial results of Q1..10 ---------- 142
Fig. 8.14 – ONE-P and TPCH-P speedup ---------- 143
Fig. 8.15 – %time increase for Q5 with different selectivity ---------- 144
Fig. 8.16 – Q5 node query variability ---------- 145
Fig. 8.17 – Total exchange of the partial results (t_{tpr}) for queries Q1..Q10 ---------- 146
Fig. 8.18 – Partial execution time of TPCH per query in each node ---------- 147
Fig. 8.20 – TEEPA partial execution times (10-node setup) ---------- 148
Fig. 8.21 – TEEPA partial execution times (30-node setup) ---------- 149
Fig. 8.22 – Average execution time for varying query loads (lower is better) ---------- 150
Fig. 8.23 – Overhead per query in the average execution time (lower is better) ---------- 151
Fig. 8.24 – Throughput for varying query loads (higher is better) ---------- 152
Fig. 8.25 – Throughput for varying query loads with a) SF =1 and b) SF =10 (higher is better) ---------- 153
Fig. 8.26 – Impact in throughput of a 10x increase in data volume (lower is better) ---------- 153
Fig. 8.27 – Influence of the query workload pattern in average execution time ---------- 154
Fig. 8.28 – Total predicate evaluation time ---------- 155
Fig. 8.29 – Number of tuple evaluations ---------- 156
Fig. 8.30 – Average execution time ---------- 157

List of Tables

Table 4.1 – Storage space required by each schema organization -- 53
Table 8.1 – Number of rows and estimate size of the TPC-H tables -- 129
Table 8.2 – Relations needed by each query -- 130
Table 8.3 – Storage space required by each schema organization -- 132
Table 8.4 – Storage space required by each schema organization for SF100 (in GB) ------------------------- 133

List of Acronyms

BI	Business Intelligence
COTS	Commercial Off-The-Shelf
CPU	Central Processing Unit
DB	Database
DBA	Database Administrator
DBMS	Database Management System
DFS	Distributed File System
ETL	Extract, Transform, Load
FIFO	First In First Out
FK	Foreign Key
GB	Gigabyte
I/O	Input/output
I/O	Input/output
ISO	International Organization for Standardization
IT	Information Technology
JDBC	Java Database Connectivity
JDK	Java Development Kit
LOC	Lines of Code
OLAP	On-Line Analytical Processing
QoS	Quality of Service
RAID	Redundant Array of Inexpensive Disks
RAM	Random Access Memory
RDBMS	Relational Database Management System
RISC	Reduced Instruction Set Computer
RLE	Run-Length Encoding
SQL	Structured Query Language
SSD	Solid-State Drive

Chapter 1

Introduction

Data Warehouses (DW) are large repositories of historical business data especially organized for reporting and analysis, and to gather insightful knowledge from the past. For that reason, DWs are becoming a fundamental tool for the decision-making process in every modern organization, especially for data-intensive organizations.

Data volumes produced by data intensive industries in competitive markets, such as telecom and smart-grids, stress the limits of DW systems. DBAs of such systems have to maintain a constant supervision of the query load, query patterns and selectivity to ensure performance. Moreover, in some industries it is also common for the results of business analysis to be feedback as inputs of operational business decision processes.

The ability to provide timely results is not just a performance issue (high throughout), but also a matter of returning query results when expected for the business decision-making process. Queries may have different time requirements according to the nature of the analysis and the business decisions. For instance, some decisions may require that the results of recent data be available in a small time frame (seconds or couple of minutes), while more strategic data analyses that span a larger temporal scope and data volume (e.g. historical data used as baseline data) may have wider time requirements (e.g. hours or days). But all are relevant, in their own way, to the business processes. The ability to deliver timely results to queries is gaining increasing interest. Pre-computation strategies can provide faster results, but their usability is limited to the

well-know (planned) queries and need to be periodically refreshed, typically after each periodic load.

On the other hand, there is also an increasing need for some data analysis to be performed over low-latency data (near real-time data) in order to more rapidly react to business and market changes. The traditional overnight periodic data load is unacceptable to some kinds of systems, like operational BI, which require more frequent data loads, preferably near real-time. This also constrains the ability of DW systems to provide timely results, even for planned queries, because of all the refreshing costs.

Parallel architectures are used to handle such increasing data volumes and to provide improved performance, by dividing both data and processing among nodes. However, they introduce several issues related to data placement, network bandwidth and parallel query processing that limits the scalability of such architectures. Moreover, they only focus in improving query performance and not in providing timely query results. Dimensioning the parallel architecture' size - the number of processing nodes - to be able to deliver timely query results is a relevant issue that is not satisfactorily answered.

1.1 Problem Statement and Research Goals

Most Data Warehouses are organized following a star schema model, commonly accepted for the last decades as the *de facto* data model in Relational Database Management Systems (RDBMS) for those applications. In this model, business performance metrics (facts) are stored in a central table (the fact table) and all relevant business perspective attributes (dimension data) are organized into a set of surrounding tables (dimension tables). Fact tables also store a set of foreign keys that reference the dimensions, and usually contain a large number of tuples. This organization of the star schema avoids redundancy. This is important, since fact tables represent a large proportion of the overall star schema size. On the other hand, dimension tables, which are usually significantly smaller in size but not in width, are stored as de-normalized relations for performance purposes [Kimball et al. 2008].

Since DWs store historical business data, they are continuously growing in size, particularly the central fact tables that store the data measures being produced by

operational systems, stressing the limits of the RDBMS, and thus resulting in increased query execution times.

Parallel architectures are used to handle such increase in data volume and provide improved performance, by dividing data and processing among nodes, usually following a shared-nothing organization. One possibility is for fact tables to be partitioned among nodes and the surrounding dimensions replicated, so that each node can independently compute partial results. However, parallel system scalability is constrained, not only by the network costs related to the exchange of temporary results between nodes, but also by the star schema model. As only the fact table is partitioned among nodes and dimensions are replicated, when the size of dimension(s) is not small, the fact that they are copied into nodes instead of partitioned may severely limit the parallelization advantage. Therefore, adding more nodes to the parallel architecture will result in limited performance improvement, and thus in sub-linear speedup.

Modern competitive markets also require results to be available in a timely fashion for helping the decision-taking process. It is not just a matter of delivering fast results, but also of guaranteeing that they will be available before business-decisions are made. While pre-computation strategies can help in providing faster results, their usability is limited to the well-know (planned) queries. In contrast, ad-hoc queries are executed without any time execution guarantees. Moreover, different levels of data processing analysis are concurrently being carried out with different aggregation levels and time-bound constraints.

In spite of all the time and effort to come up with a parallel infrastructure to handle such increase in data volume and to improve query execution time, it may be insufficient to provide timely query execution, particularly for ad-hoc queries. The performance of well-known queries can be tuned through a set of auxiliary strategies and mechanisms, such as materialized views and index tuning. However, for ad-hoc queries, such mechanisms are not an alternative solution. The query patterns unpredictability result in unpredictable query execution times, which may be incompatible with business requirements. Moreover, the query execution time is also influenced by the query load that is being run simultaneously, each competing for resources (IO, CPU, memory,...).

Given these issues, we formulate the aims of this dissertation as:

How to provide scalable, predictable and timely results for DW queries with large data volumes and concurrent query workloads? How to dimension the parallel architecture to provide timely execution even when large query loads are being executed simultaneously? How to provide unlimited scalability for recent and historical data?

To address the above-discussed issues, we had to investigate new approaches for solving query execution time unpredictability and for providing timely results, and also overcoming the scalability limitations related to the star schema model. In particular, this research aimed to:

- Provide predictable execution times for all queries, both well-known (planned) and ad-hoc queries
- Propose mechanisms to overcome the scalability limitations of traditional parallel data warehouse architectures
- Propose an architecture that can handle unlimited data volume scalability, and be deployed over an elastic set of heterogeneous processing nodes
- Propose mechanisms that can provide the query execution with different timely constraints (bounded time)
- Propose mechanisms that allow a massive number of queries to be ran concurrently without significant performance degradation

1.2 Dissertation Contributions

This dissertation proposes a massively scalable parallel DW architecture that uses an elastic set of processing nodes to provide predictable and timely results. The main research contributions can be summarized as:

- **To investigate how to deliver predictable execution time, particularly for ad-hoc queries.** The approach must be able to provide:
  - predictable execution times for current and future data volumes.
  - methods for providing and expressing temporal objectives;
- **To investigate how to provide unlimited data scalability using a parallel DW approach.** The approach must be able to:
  - provide improved scale-up capabilities
  - handle huge volumes of data and be massively parallel
- **To investigate how to support massive concurrency without performance degradation and while guaranteeing target response time metrics.**
- **To investigate how to provide timely results under variable query workloads.** The approaches must be able to:
  - handle variable query loads
  - maximize and share the processing effort among concurrent queries
- **To investigate how to manage the architecture on an elastic set of heterogeneous nodes.** The approaches must be able to provide:
  - query and cost requirements
  - methods to allocate data and work according to the processing nodes characteristics.

Parts of this work are described in the following publications:

- *"Data Warehouse Processing Scale-up for Massive Concurrent Queries with SPIN", João Pedro Costa & Pedro Furtado, in TLDKS Journal, Transactions on Large-Scale Data- and Knowledge-Centered Systems, Springer 2015, ISBN 978-3-662-46334-5.*
- *"Improving the Processing of DW Star-Queries under Concurrent Query Workloads", João Pedro Costa & Pedro Furtado, In Proceedings of the 16th International Conference on Data Warehousing and Knowledge Discovery, DaWaK 2014, Munich, Germany. Springer, ISBN 978-3-319-10159-0*

- *"SPIN: Concurrent Workload Scaling over Data Warehouses", João Pedro Costa & Pedro Furtado, In Proceedings of the 15th International Conference on Data Warehousing and Knowledge Discovery, DaWaK 2013. Prague, Czech Republic. Springer 2013, ISBN 978-3-642-40130-5*
- *"Providing Timely Results with an Elastic Parallel DW", João Pedro Costa, Pedro Martins, José Cecilio, & Pedro Furtado, In Proceedings of the 20th International Conference on Foundations of Intelligent Systems, ISMIS'12, Macau, China. Springer, ISBN 978-3-642-34623-1*
- *"TEEPA: a Timely-aware Elastic Parallel Architecture", João Pedro Costa, Pedro Martins, José Cecilio, & Pedro Furtado, In Proceedings of the 16th International Database Engineering & Applications Symposium, IDEAS'12, Prague, Czech Republic. ACM 2012, ISBN 978-1-4503-1234-9.*
- *"Overcoming the Scalability Limitations of Parallel Star Schema Data Warehouses", João Pedro Costa, José Cecílio, Pedro Martins, & Pedro Furtado, In Proceedings of the 12th International Conference on Algorithms and Architectures for Parallel Processing, ICA3PP'12, Fukuoka, Japan. Springer, ISBN 978-3-642-33077-3*
- *"A Predictable Storage Model for Scalable Parallel DW", João Pedro Costa, Pedro Martins, José Cecílio, & Pedro Furtado, In Fifteenth International Database Engineering and Applications Symposium (IDEAS 2011), Lisbon, Portugal. ACM 2011, ISBN 978-1-4503-0627-0*
- *"ONE: a Predictable and Scalable DW Model", João Pedro Costa, José Cecílio, Pedro Martins, & Pedro Furtado, In Proceedings of the 13th International Conference on Data Warehousing and Knowledge Discovery, DaWaK'11, Toulouse, France. Springer, ISBN 978-3-642-23543-6*

1.3 Organization of the Dissertation

The remainder of this dissertation is divided into eight chapters:

Chapter Two: Background and Related Work. Initially we provide background concerning costs of processing queries over parallel data warehouses (PDW), and the influence of data volume and query workload on such costs. We also discuss the scalability limitations of the star schema model, and its inability to efficiently scale-up with larger data volumes.

The second part of the chapter examines related work concerning data and processing models, parallel data warehouses architectures, time-related Quality of Service approaches, and strategies for data processing sharing.

Chapter Three: A Timely and Massive Scalable DW. This chapter describes the mechanisms of the scalable architecture proposal for providing timely results for scalable data volumes, concurrent queries and near-real time requirements.

Chapter Four: A Scalable and Predictable Data Model. In this chapter, we propose a de-normalized data model, called ONE. The physical de-normalized data organization and simplified query processing approach provides predictable time guarantees and unlimited data volume scalability.

The second part of the chapter proposes a method that adapts the data model organization, and the data de-normalization level to best fit node characteristics. It also discusses a logical to physical model view that performs the necessary query rewriting and processing actions according to the data model organization.

Chapter Five: Providing Right-time with an Elastic Parallel Architecture. In this chapter we describe an approach which uses an elastic set of heterogeneous nodes to adapt the parallel infrastructure to handle dynamic amounts of data and query loads, and guarantee tight time requirements. This chapter proposes different cost-based approaches to selecting the node configuration that best fits the specified time and cost requirements.

Chapter Six: SPIN: Concurrent Workload Scaling over Data Warehouses. In this chapter, we propose the SPIN data processing model, which as opposed to the query-at-

a-time model, uses the data-at-time model. SPIN minimizes data IO costs by sharing data reads among all running queries, and therefore handles massive concurrent queries without significant performance degradation.

Chapter Seven: Providing Right-Time Guarantees to Scalable Concurrent Workloads. In this chapter, we propose mechanisms that determine if the current SPIN workload processing tree can provide a query's right-time guarantees, which for large data volumes it may result in data reading costs greater than the required right time, and propose a parallel SPIN approach, called CARROUSEL, that orchestrates several SPIN processing engines in parallel, to speedup query processing and to reduce query execution time below the required time targets.

Chapter Eight: Experimental Evaluation. This chapter evaluates the proposed mechanisms. It presents some experimental results that demonstrate the ability of the proposed architecture to scale over large data volumes, to scale over high numbers of concurrent queries and guarantee timings, even with massive data amounts.

Chapter Nine: Conclusions and Future Work. This chapter concludes this dissertation by presenting a summary of the key contributions, and presents some final considerations and finishes with a discussion of open aspects left for future work.

Chapter 2

Background and Related Work

In this work, we propose strategies to provide timely execution of DW queries for scalable data volumes and under large concurrent query loads. This chapter starts by presenting some background introduction to Data Warehouses (DW), including storage models and query processing, and their influence in the execution times, which is unpredictable, particularly for ad-hoc queries (non-planned queries).

In the following sections, we review related work on data de-normalization and decomposition, and approaches in data sharing and pipeline processing.

In the next section we overview different parallel architectures, and discuss how parallel query processing, the data volume, data partitioning and allocation issues influence the query execution time. We discuss the scalability limitations of the star schema model, and its inability to efficiently scale-up with larger data volumes. In the last section we examine related works on providing predictable execution and strategies for data processing sharing.

2.1 Data Warehouse Background

Data warehousing emerged in the nineties, aiming to support organizations in decision-making process. Management demanded the ability to make business analyses with the data gathered by operational systems. However, these systems were designed for business operations purposes and were complex and not well suited for processing business analyses.

Data Warehouses (DW) are large repositories that store large amounts of past historical business data, structured and organized to deliver business analyses and thus

support the decision making processes. Kimball [Kimball 1996] defines a Data Warehouse as "*… a copy of transaction data specifically structured for query and analysis*". Fig. 2.1 depicts the data warehousing environment.

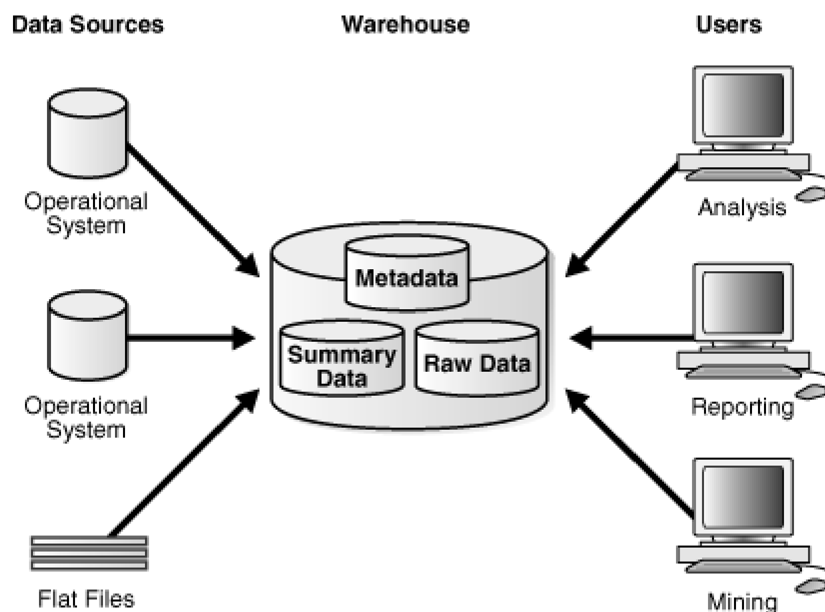


Fig. 2.1 – The Data Warehouse environment (figure extracted from [Strohm 2011])

Data produced by operational systems, must be **E**xtracted from operational and external data sources an integrated into a common staging area for **T**ransformation and cleansing before being **L**oaded into the DW and made available for business analyses. This process is commonly known as the ETL process. The ETL process can be complex and resource demanding and therefore usually it is performed at given time intervals (every day or hour) to batch processing the new data produced between loads.

2.1.1 Data Warehouse models

With the advent of Data Warehouses, two distinct data models were proposed to store DW base data: the star schema model and the snowflake model.

The star schema model [Kimball 1996] organizes the base data, for instance the sales data, in a star fashion schema. A central fact table contains the business metrics (e.g. unit sold, unit price) and is surrounded by a set of dimension tables representing the different business perspectives (e.g. by product, by customer, by store). Fig. 2.2 illustrates a typical sales star schema.

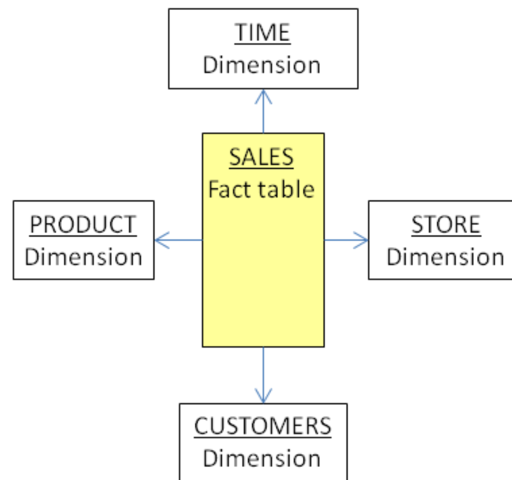


Fig. 2.2 – An example of a sales star schema model

The fact table is highly normalized, containing a set of foreign keys that reference the surrounding dimension tables, and stores the measure facts. The physical division in normalized fact tables (with metrics) and de-normalized dimension tables allows a trade-off between performance and storage space. It also offers a simple business understanding of the model, composed by a set of metrics (facts) and attributes for business analysis (dimensions). The central fact table is highly normalized in order to minimize data redundancy and storage space. Usually, the fact table represents a large percentage of the overall storage space. On the other hand, dimension tables are highly de-normalized and represent a smaller fraction of the overall DW storage space. The star schema model has becoming the *de facto* data model followed in most DW deployments in relational database management systems (RDBMS), because of its simplicity and ease of understanding by non-IT users.

The snowflake model is related to the star schema model, but normalizes (splits) some of the dimension tables, particularly the wider dimensions, to reduce the level of data redundancy.

2.1.2 Typical aggregation queries

The query workload of a DW is typically composed by star-join aggregation queries, accessing the fact and dimension tables. Dimension attributes are used as filtering conditions (WHERE clause) and as grouping attributes, and aggregation functions are applied to fact attributes (measures). Each dimension table is joined with the fact table using a primary to foreign key join. Fig. 2.3 illustrates such type of queries, which typically returns a small number of rows as a result set.

```
SELECT     dim attributes, aggregation functions
FROM       fact, set of dimension tables
WHERE      join conditions
AND        dim attribute conditions
GROUP BY   dim attributes
```

Fig. 2.3 – Template of typical aggregation query

Since a DW stores historical business data and is periodically loaded with new data gathered from operational systems, it is continuously growing in size, particularly the central fact table that stores the data measures of new events produced by operational systems. Due to its nature, a fact table is usually only subject to insert operations, while the same doesn't necessary happens to the dimension tables. Inserts also occur, but at lower rates when compared with the fact table. Dimension tables are also subject to update operations.

An overview of data warehousing is presented in [Chaudhuri & Dayal 1997; Kimball et al. 2008] and a survey on logical and physical design issues of Data Warehouses is presented in [Golfarelli & Rizzi 2009].

2.2 Query processing

For each query, the database system builds a query execution plan containing all the necessary processing steps to produce the query results. This execution plan is query-specific, and depends of several factors, such as the query predicates, the number and size of the required tables, how they are joined and the join order, the table selectivity and the access method used. Fig. 2.4 shows an example of an execution plan.

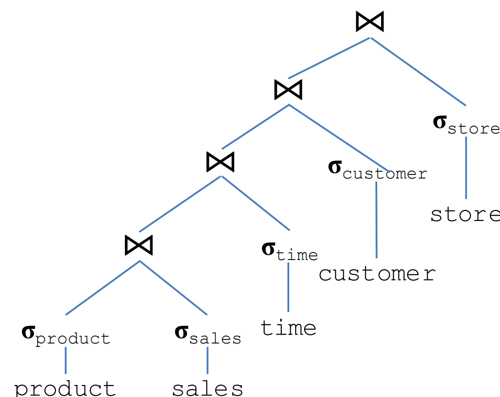


Fig. 2.4 – Example execution plan

The query execution time involves the cost of joining all the required relations (fact and dimensions). For large relations this result in costly tasks and therefore in poor query execution times. It is also hard to predict execution times since they are highly influenced by each relation's size, selectivity and number of involved relations. The influence of these unpredictability factors is higher as the data volume increases.

2.2.1 Join processing

Over the past decades, significant effort has been made to develop efficient join algorithms. Nested-loop and sort-merge [Blasgen & Eswaran 1977] are the two most common ones that can be found in most databases engines. The former is used for small relations and the latter used for large relations. The grace hash join [Kitsuregawa et al. 1983] and the hybrid hash join [Zeller & Gray 1990] algorithms apply a hash function to relations to speed up the join process. The evaluation costs of the algorithms were revisited in [Patel et al. 1994] and [Harris & Ramamohanarao 1996]. More recently [Kim et al. 2009] revised the algorithms on modern multi-core CPUs.

DeWitt et al [DeWitt et al. 1984] compared the sort-merge, simple hash, grace hash and hybrid hash join algorithms and concluded that hybrid hash has the lowest cost. Graefe et al. [Graefe et al. 1994] concluded that both sort merge and hash join should be available in any database system and the join algorithm to be used should be chosen by a query optimizer based on the input data.

The G-Join algorithm [Graefe 2011] combines elements of the three traditional join algorithms, acting like a merge join in the case of two sorted inputs and like hash join in the case of two unsorted inputs, including taking advantage of different input sizes.

Surveys of join algorithms appear in [Mishra & Eich 1992; Graefe 1993; Graefe et al. 1994; Kim et al. 2009; Graefe 2011].

Analysis: The join processing cost is highly influenced by factors such as the data inputs, the join order, and query selectivity. Since our goal is to provide predictable query execution times to large data warehouses, we remove these unpredictability factors by using a *"no-join"* approach, where all data attributes are stored in a single relation.

2.2.2 Mechanisms for improving query performance

The query execution time is also influenced by the access method used to gather the relevant data, e.g. full sequential scan, indexes. The types of indexes available on most databases engines are B-Tree index [Bayer & McCreight 1970; Comer 1979] and Bitmap index [O'Neil & Quass 1997; Johnson 1999].

A B-Tree is a self balancing search tree that maintains attribute values in a sorted order, together with a reference to its position in the table. However, since each query frequently involves a different set of attributes, a separate index for nearly every combination of attributes is required to deliver top performance to all the queries. As the number of indices grows, the associated storage space requirements increase exponentially with the number of attributes.

Bitmap indexes use each distinct value of the indexed attribute as a key, and generate a bitmap containing as many bits as the number of records in the data set for each key. For low-cardinality attributes, it requires less space and provides performance advantage by performing bitwise logical operations on the bitmaps. An evaluation of bitmap index is given in [Chan & Ioannidis 1998; O'Neil et al. 2007]. [Stockinger & Wu 2006] review various bitmap index technologies, including bitmap encoding, compression and binning, in data warehouse applications.

Bitmap join indexes [O'Neil & Graefe 1995] are very efficient materialized structures for avoiding costly joins. A bitmap indicates which fact rows correspond to each attribute value of a dimension table and represents a pre-computed result of a join between the fact and a dimension table.

A materialized view [Roussopoulos 1998] physically stores (materializes) all the rows of a view in storage system, to provide fast answers to queries that follow the view pattern. It is particularly useful to frequent, well known and planned queries. After each periodic load, the materialized results have to be refreshed in order to maintain data consistency. The maintenance costs and the storage limitations restrict the number of materialized views.

The problem of selecting the views to materialize in order to optimize the total query response time under a disk-space constraint is studied in [Gupta 2000]. A more flexible materialization strategy to reduce the storage space and view maintenance costs

is proposed in [Zhou et al. 2007], which selectively materializes only a subset of rows, for example, the most frequently accessed rows. A survey on selection of views is presented in [Mami & Bellahsene 2012].

[Abiteboul & Duschka 1998] show that the complexity of answering queries using materialized views depends on whether views are assumed to store all the tuples that satisfy the view definition, or only a subset of it. An experimental evaluation of an automated selection of materialized views and indexes is presented in [Agrawal et al. 2000].

Although some queries submitted to a DW may be known in advance, and therefore can be materialized, most queries are *ad-hoc*. Sampling [Acharya et al. 1999] can provide fast approximate results to all aggregated queries. Sampling trades-off precision for performance by employing the power offered by statistical methods to reduce the data volume that has to be processed to compute an acceptable result. However, the sampling size and the lack of data samples of some aggregation groups limit the execution time and results precision. To overcome this issue, a minimal group representation sampling is proposed in [Acharya et al. 2000] and a time-based stratified sampling is proposed in [Costa & Furtado 2003]. [Jermaine et al. 2004] proposed an online maintenance of very large random samples.

Analysis: DW deployments extensively use both indexes and materialized views to provide faster execution times to some query patterns. But they do not provide predictable execution times and the additional storage space and maintenance costs can be very large. Although these structures can be combined with our proposals, we aim to provide predictable performance by extensively promoting data and processing sharing among queries. While the storage requirements of our *"no-join"* data model (called ONE), is higher than the star schema model, these difference decreases when the storage size of these auxiliaries structures is also accounted.

2.3 De-normalization and decomposition

Our proposed model (ONE), follows a de-normalized approach and physically stores all the star schema data in a single relation, i.e. the fact table also contains the related dimension attributes, while providing a conceptual star schema logical view.

The first research works on schema de-normalization where proposed in the context of universal relations [Fagin et al. 1982; Korth et al. 1984]. A methodology to assess the de-normalization effects using relational algebra operations and query trees is presented in [Sanders & Shin 2001].

The use of hierarchical de-normalization as a possibility to optimize the data warehouse design is evaluated in [Zaker et al. 2008; Zaker et al. 2009]. A framework for systematic database de-normalization is proposed in [Pinto 2009]. However these works do not focus on the de-normalization benefits of the star schema model, and they do not offer a clear insight of the query performance predictability.

The decomposition data model was first discussed in [Copeland & Khoshafian 1985]. More recently [Stonebraker et al. 2005; Zhang et al. 2010] proved that vertical partitioning and column-wise store engines are effective in reducing the disk IO and thus boosting query performance. However, these works focus on improving query performance and minimize the cost of joining DW relations using a star schema model, and not on providing a predictable and time invariant execution time environment.

Blink [Raman et al. 2008] partitions data by frequency to achieve good compression while maintaining long runs of fixed length codes, however is unable to efficiently handle updates against this compressed data format, requiring huge processing costs to reorganize and repartition data. The implementation and performance of compressed databases is studied in [Westmann et al. 1998; Westmann et al. 2000; Chen et al. 2001; O'Connell & Winterbottom 2003; Poess & Potapov 2003; Holloway et al. 2007].

Analysis: We also use de-normalization as a way to increase query performance. We de-normalization the star schema (fact and dimensions tables) provide a conceptual star schema logical view and propose mechanisms for predictable execution time.

Recently, WideTable [Li & Patel 2014] extended the ONE de-normalized model to a subset of the dimensions, thus converting complex queries into simple scans, and demonstrated its effectiveness in terms of raw performance and scalability.

2.4 Data sharing and Pipeline Processing

When analyzing a query execution plan, we observe that the low-level data access methods, such as sequential scan, represent a major weight in the total query execution time. One way to reduce such a burden is to store relations in memory. However, the amount of available memory is limited and may be insufficient to hold large DW, and it is also required for performing join and sort operations.

Cooperative scans [Zukowski et al. 2007] enhance performance by improving data sharing between concurrent queries. It performs a dynamic scheduling of queries and their data requests taking into account the current executing actions. A query execution can be postponed, waiting for similar queries to arrive in order to share the IO cost. While this minimizes the overall IO costs, by mainly using sequential scans instead of a large number of costly random IO operations, and the number of scan operations (since scans are shared between queries), it introduces undesirable delays to query execution and does not deliver predictable query execution times.

QPipe [Harizopoulos et al. 2005] applies on-demand simultaneous pipelining of common intermediate results across queries, avoiding costly materializations and improving performance when compared to tuple-by-tuple evaluation. Each operator is promoted to an independent micro-engine, called μEngine, which accepts requests and serves them as queues. It introduced the concept of “Window of opportunity”, as the time interval where newly submitted operators can take advantage of the one already in progress. Resource utilization is improved when requests of the same nature are grouped together, and when dedicated processes are used to process each group of similar requests.

CJoin [Candea et al. 2009; Candea et al. 2011] applies a continuous scan model to the fact table, reading and placing fact tuples in a pipeline, and sharing dimension join tasks among queries, by attaching a bitmap tag to each fact tuple, one bit for each query, and attaching a similar bitmap tag to each dimension tuple referenced by at least one of the running queries. Each fact tuple in the pipeline goes through a set of filters (one for each dimension). A bitwise operation is performed against the dimension bitmap of the corresponding tuples to determine if it is referenced by at least one of the running queries. If not, the tuple is discarded. Tuples that reach the end of the pipeline

(tuples not discarded in filters) are then distributed to dedicated query aggregation operators, one for each query.

CJoin schedules processing tasks so that they share IO, particularly scanning tasks. However it requires that each dimension table be in memory so that it can probed to perform hash joins, and to continuously update the dimension bit vectors (with varying numbers of bits) when a new query is submitted or running queries finishes. However, if the size of a dimension is large, it may require the execution of an external hash-join, resulting in slower performance and unpredictable query execution time. [Candea et al. 2011] did not evaluate the impact of selectively adding tuples to dimension hash tables, and concurrency issues related to registering and de-registering queries and the bitwise computation. An overhead analysis is required, regarding the necessary memory and computational costs for maintaining the bit vector when the number of concurrent queries increases.

Crescando [Unterbrunner et al. 2009] is based on parallel and collaborative scans in main memory and the so-called "query-data" joins known from data-stream processing. Crescando loads a tuple into memory and then “joins” the tuple with all interested queries, so that the cost associated with loading the tuple into memory is amortized.

DataPath [Arumugam et al. 2010] is a “data-centric” system where queries do not request data, instead the data is automatically pushed onto processors. It resembles the QPipe and the main-memory-based Crescando system in the way it attempts to share memory access latency and bandwidth.

SharedDB [Giannikis et al. 2012] introduce the concept of Global Query Plans, which compiles a single plan for the whole workload, instead of compiling each individual query into separate plans. This plan serves multiple concurrent queries and may be reused over a long period of time. The proposed Shared Join plans approach, which combines (union) relation tuples of all concurrent queries before performing a single large shared join, instead of multiple smaller joins, only proved to be efficient for large number of concurrent queries. SharedDB batches queries and updates and thereby makes use of traditional, best-of-breed algorithms to implement joins, sorting, and grouping. While one batch of queries and updates is processed, newly arriving queries and updates are queued. When the current batch has been processed, a new batch is

created with the queued queries. This batch-based execution model adds latency to each query.

Analysis: Our proposal shares some similar characteristics with those solutions, namely the data sharing, the data pipeline processing, and processing and sharing data scans in a circular loop. While SharedDB uses standard query processing techniques such as index nested-loops, hashing and sorting for any kind of operator of the relational algebra (e.g., joins, grouping, ranking, and sorting), CJoin and DataPath are limited to sharing the join computation and to the cases in which the particular CJoin and DataPath join methods show good performance.

We tackle the dimension size problem using a different approach, which has smaller memory requirements and can effectively be deployed into a wide range of parallel shared nothing architectures composed of heterogeneous processing nodes. Our SPIN proposal is conceptually related to CJoin, and QPipe in what concerns the continuous scanning of fact data, but it uses a less complex approach and does not have the memory limitations of such approaches. SPIN uses a de-normalized model, as proposed in [Costa, Cecílio, et al. 2011] as a way to avoid the join costs, at the expense of additional storage cost. Since it provides full data and processing scalability, it allows massive parallelization [Costa, Martins, et al. 2011], provides balanced data distribution, scalable performance and predictable query execution times. However, the CJoin logic for small in-memory relations and the dynamic scheduling of cooperative scans can be integrated in SPIN.

2.5 Parallel DW, Data Partitioning and Placement

A survey of parallel and distributed DW is given in [Furtado 2009a]. Since DW store historical business data, they are continuously growing in size, stressing the limits of the database systems and thus resulting in increased query execution times. Parallel architectures are used to handle such increase in data volume and provide improved performance, by dividing the data and processing among nodes. Three basic parallel computer architectures are presented in [Dewitt & Gray 1992; Ozsu & Valduriez 2011], illustrated in fig. 2.5, depending on how main memory or disk is shared: shared-memory, shared-disk and shared-nothing.

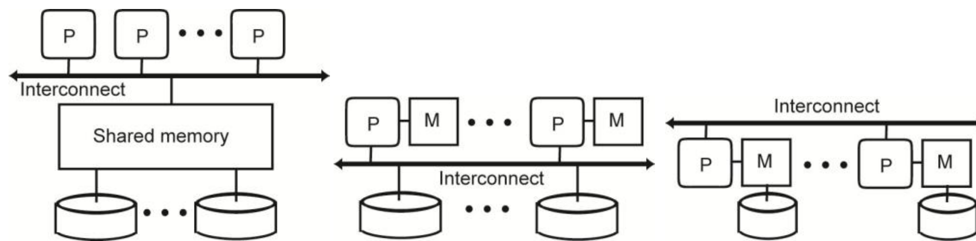


Fig. 2.5 – Parallel approaches a) shared-memory b) shared-disk c) shared-nothing (figure extracted from [Dewitt & Gray 1992])

Each as its merits, but in what concerns data scalability the shared-nothing approach minimizes interference by minimizing resource sharing [Dewitt & Gray 1992]. To minimize network exchange costs for processing joins in a shared-nothing approach and maximize the local computation of partial results, the data is usually distributed using a Partition and Replicate Strategy (PRS) [Epstein et al. 1978; Copeland et al. 1988; Yu et al. 1989; Liu & Yu 1992; Baru et al. 1995; Noaman & Barker 1999], where the larger relation is partitioned and the others are replicated. In parallel shared-nothing DWs, fact tables are partitioned into smaller partitions and allocated to nodes, and dimensions are frequently fully replicated into each node (regardless of their size), as illustrated in Fig. 2.6. DWS [Bernardino 2002] replicates dimensions and uses a uniform data striping approach to evenly divide the fact data among processing nodes, which allows it to deliver approximate query results when one or more nodes fail.

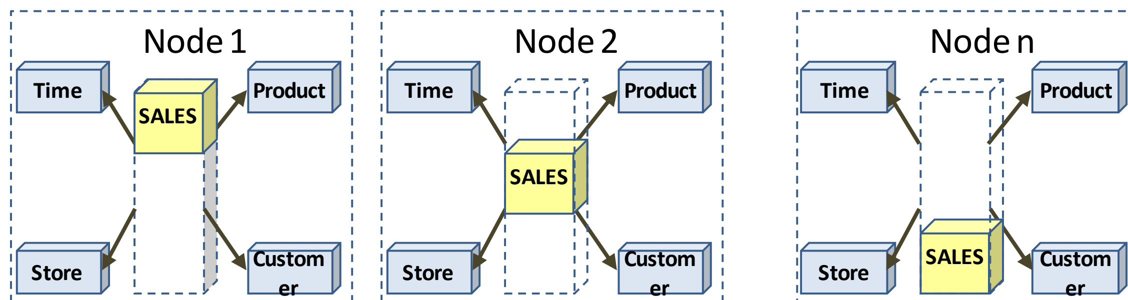


Fig. 2.6 – A parallel deployment of a star schema with replicated dimensions

With this data distribution scheme, each node independently computes its partial results locally, as shown in Fig. 2.7. Parallel database systems split the processing of a query in a set of steps [Dewitt et al. 1990; Shatdal & Naughton 1994; Graefe 1993; Jaedicke & Mitschang 1998; Bernardino 2002]. A query (1) received by a submitter node is rewritten (2) and forwarded to the processing nodes (3). Each processing node executes the rewritten query against the local data (4) before sending the partial results (5) to the merger node. The merger node, which may be the submitter node, waits for

the intermediate results, and merges them (6) to compute the final query result, before sending it (7) to the user.

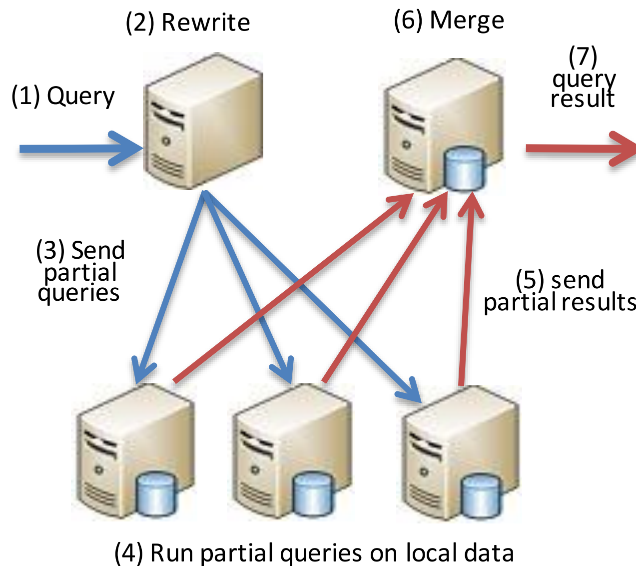


Fig. 2.7 – Query processing in shared-nothing

The overall query execution time is mostly influenced by the local query execution of the slowest node, the number of nodes, the partial results' size and the cost of sending them to the merger node. The local query execution time can be improved by reducing the amount of data allocated to each node.

When dimensions are replicated, their relative weight in each node local storage space increases with the number of nodes. As a consequence, we observe smaller decrease in the local query processing time to a point that adding more nodes represents a minimal local performance improvement. [Costa & Madeira 2004] proposed a selective loading strategy to deal with data warehouses with big dimensions. Equi-partitioning may help in this matter, by partitioning both the fact table and some large dimension on a common attribute, usually the dimension primary key, and allocating related partitions into the same node. [Bellatreche et al. 2000] proposed an algorithm that selects dimensional tables for fragmentation and then fragment fact table based on these dimensional tables. [Shasha & Wang 1991; Liu & Chen 1996] explored horizontal fragmentation and hash partitioning of relations and intermediate results to perform parallel multiway join queries and increase the parallelism. [Stöhr et al. 2000] developed a multi-dimensional hierarchical data fragmentation and allocation method to partition the fact table and distribute among nodes. Data placement issues in shared-

nothing parallel architectures are discussed in [Copeland et al. 1988; Achyutuni et al. 1995; Mehta & DeWitt 1997; Cheung et al. 2012; Bellatreche et al. 2000].

[Furtado 2004] exploits a workload-based data placement and join processing in parallel DW. A map-reduce-like approach was explored in [Furtado 2008; Furtado 2009b; Yang et al. 2010] to overcome this load unbalance, by partitioning data into a large number of small data chunks (greater than the available nodes), with some being replicated for dependability or performance reasons. Chunks are processed as nodes become available to process new data. However, it does not solve the increasing weight of dimensions and results in higher network costs, since more partial results (one for each chunk) have to be exchanged. While this overcomes the storage scalability limitations, it introduces additional query complexity related to processing parallel network joins, and is sensitive to network costs related to exchanging intermediate results. The parallel nested loop (PNL) algorithm, the parallel associative join (PAJ) algorithm, and the parallel hash join (PHJ) algorithm are presented in [Ozsu & Valduriez 2011, chap. 14]. A performance evaluation of parallel joins is presented in [Schneider & Dewitt 1989], and a thorough discussion of how parallel and distributed DW work is presented in [Furtado 2009a].

Frequently, large non-equi-partitioned dimensions are also partitioned and distributed among nodes, without being co-located according to the fact table data. However, since business data inherently do not follow a random distribution, and data is skewed, equi-partitioning may introduce another limitation to scalability, with some nodes storing more data than others. Research in handling skew in parallel joins include [DeWitt et al. 1992; Xu et al. 2008; Xu & Kostamaa 2009; Wu & Madden 2011]. A skew-aware automatic database partitioning scheme is proposed in [Pavlo et al. 2012].

Cyclo-join [Frey et al. 2009; Frey et al. 2010] is a distributed join processing approach that leverages the potential of RDMA-enabled hardware provided by modern high-throughput networks. It organizes the hosts in a circular ring and reduces the amount of data that is shipped over the network, by only forwarding the data partition to the next host.

The scalability of parallel architectures is thus constrained by those correlated factors: the local data processing, data unbalance among nodes, in-network parallel joins, the number of nodes, and the number of partitions per node. Two main paradigms

are used to attain that goal: parallel DBMS and Map-Reduce (MR) frameworks. Pavlo et al. [Pavlo et al. 2009] presents a comparison of approaches to large-scale data analytics, identifying their limitations both in performance and usability. While MR offers elastic scalability, it lacks the expressiveness and usability of SQL. A hybrid system that approaches parallel databases in performance and efficiency, yet still yields the scalability, fault tolerance, and flexibility of MapReduce-based systems is explored in [Abouzeid et al. 2009; Xu et al. 2010]. Hive [Thusoo et al. 2009] is an open-source data warehousing solution built on top of Hadoop that compiles queries expressed in SQL-like language (HiveQL) into hadoop map-reduce jobs.

Both paradigms show scalability limitations in processing in-network joins and in providing timely executions. As a consequence, the traditional approach of adding more nodes to the parallel infrastructures is insufficient to provide timely guarantees, because of their inability to provide scale-up execution guarantees. In spite of all the effort to devise improved parallel joins algorithms, this network scalability limitation introduced by parallel joins is not solved.

Analysis: Our proposals also consider a shared-nothing environment. We advocate that, to achieve massive scalability and provide timely results, parallel joins have to be removed. Our proposed data model (ONE) is skew-free and can be extensively partitioned and deployed into an elastic set of heterogeneous nodes to provide execution time guarantees. We proposed a data placement algorithm that uses the time targets and each node characteristics, to determine the data volume allocated to each node.

2.6 Predictable execution time

The ability to provide predictable execution time and query results in a timely manner is gaining increasing importance, as more and more operational decisions are being made using DW data analyses. It is not just a matter of providing fast answers, but also of guaranteeing that the answers will be there (available) when expected and needed.

Main-memory solutions [Kallman et al. 2008; Huang et al. 2009; Grund et al. 2010] enhance query performance by removing the associated IO reading costs. However, since memory is limited and the DW volume is continuously increasing in size, the available memory may be insufficient to process join and sort operations and also to maintain the entire DW in memory.

Blink [Raman et al. 2008] is an in-memory based query processor that heavily exploits the underlying CPU infrastructure. It uses a frequency partitioning scheme, to provide good lossless compression, which produces long runs of fixed length codes and thus provide constant time decoding. However, it is unable to efficiently handle periodic data loads with this compressed data format. It requires huge processing costs to reorganize and repartition data in order to maintain order-preserving codes. This order-preserving is essential to allow the evaluation of equality and range predicates to coded values.

Crescando [Unterbrunner et al. 2009] attempts to share the memory access latency and bandwidth among queries. Crescando loads a tuple into memory and then “joins” the tuple with all interested queries, so that the cost associated with loading the tuple into memory is amortized. While the proposed approach is not always optimal for a given workload, it provides latency and freshness guarantees for all workloads.

CJoin [Candea et al. 2009; Candea et al. 2011] improves throughput in large-scale data analytics systems processing many concurrent join queries. It employs a single physical plan that shares I/O, computation, and tuple storage across instead of using the conventional query-at-a-time model. The predictable performance is obtained by employing a sequential scan to the fact table and performing joins with in-memory dimensions. However, dimensions could be large and may not fit in memory. Predictable performance can only be given for star-join queries.

SharedDB [Giannikis et al. 2012] batches queries and updates, thereby making use of traditional, best-of-breed algorithms to implement joins, sorting, and grouping. While one batch of queries and updates is processed, newly arriving queries and updates are queued. When the current batch has been processed, a new batch is created with the queued queries and then the queue is emptied. This batch-based execution model adds latency to each query. A specific advantage of SharedDB as compared to QPipe and DataPath is its ability to meet SLAs and bound the response time of queries.

Analysis: Our work distinguishes from these works in the following aspects: we do not have joins, therefore we have minimum memory requirements, while SharedDB and CJoin have large memory requirements to hold dimensions and to perform in-memory joins; we can handle small and large dimension sizes, while CJoin and Blink can only provide predictable performance to dimensions that can fit in-memory; we can provide predictable performance to data residing in memory and our model does not suffer from data skew. We also proposed a right-time manager that takes advantage of the minimum processing instruction set of our approach, to balance the data load among an elastic set of heterogeneous processing nodes.

Chapter 3

A Timely and Massively Scalable DW

Data Warehouse systems are a fundamental tool for the decision-making process. Today, they have to deal with increasingly large data volumes, which is typically stored as a star schema. The query workload is also more demanding, involving more complex, ad-hoc and unpredictable query patterns, with more simultaneous queries being submitted and executed concurrently. Fig. 3.1 illustrates the unpredictability factors that influence the ability to provide timely results to queries.

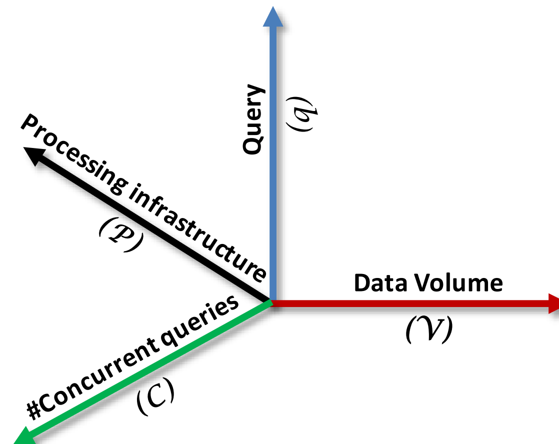


Fig. 3.1 – Unpredictability factors that influence the ability to provide timely results

Several factors influence the ability of the DW infrastructure to provide timely results to queries, such as the query (query selectivity, number of relations that have to be joined, the join algorithms and the relations sizes), the heterogeneity and capabilities of the processing infrastructure *(P)*, including IO throughput, the memory available to process joins, and the implementation of the join algorithms. The data volume *(V)* and the concurrent query load *(C)*, number of concurrent queries that are executing simultaneously, also influence the system ability to provide predictable execution times.

Definition 3.1

For a star schema model S composed of {F, D}, where F denotes the fact relation and $D=\{d_1,d_2, ..., d_n\}$ denotes the set of dimensions, with d ranging on D. Let $Q = \{q_1,q_2, ..., q_n\}$ denotes a set of aggregation queries, with q ranging on Q, submitted to {F, D'} with D'⊆D. Let V be the data volume, C the number of queries that are currently being executed and P the processing infrastructure. The query execution time of q, $t_{exec}(q)$, is

$$t_{exec}(q) = f(a_q, \mathrm{V}, \mathrm{P}, \mathrm{C})$$

where $a_q = \pi(\sigma_q(\mathrm{F} \bowtie \mathrm{D'}))$, *represents the relational algebra representation of the query q (for simplicity of the formula we did not include aggregation, which can be added).*

This chapter describes the data warehousing architecture proposed in this dissertation, which aims to provide scalability and timely results for massive data volumes. The architecture is able to do this even in the presence of a large number of concurrent queries and is able to meet near real-time requirements. The ability to provide timely results is not just a performance issue (high throughput), but also a matter of returning query results when expected, according to the nature of the analysis and the business decisions.

The chapter starts by proposing mechanisms to provide predictable execution and unlimited data scalability. The concept of timely data analysis (right-time execution) is then introduced, and we propose mechanisms to provide right-time guarantees while meeting runtime predictability and freshness requirements. Finally we present the mechanisms of the architecture that allow it to still guarantee right-time execution in the presence of huge concurrent query loads.

3.1 Providing predictable execution

For a query *q* submitted to a particular processing infrastructure *p* with a given data volume *v*, the execution time of q, $t_{exec}(q) = f(a_q, v, p, 1)$, is mostly influenced by a_q, the query selectivity, the number of relations that have to be joined together, the memory available for joins, the used join algorithms and the join order.

The query execution complexity is highly influenced by the number of relations that have to be joined together (D'), the relations' size and the query selection

predicates (selectivity), which influences the data volume that has to be read from storage and joined. This data volume and the memory available for joins, influence both the join order and the used join algorithms. These unpredictable costs related to joining the fact table with dimensions relations (F ⋈ D') arise from the star schema organization.

To provide predictable execution times, we propose the ONE data model, where the fact table and the corresponding dimensions are physically stored into a single de-normalized relation, without primary and foreign keys, containing all the attributes from both fact and dimension tables, as illustrated in Fig. 3.2.

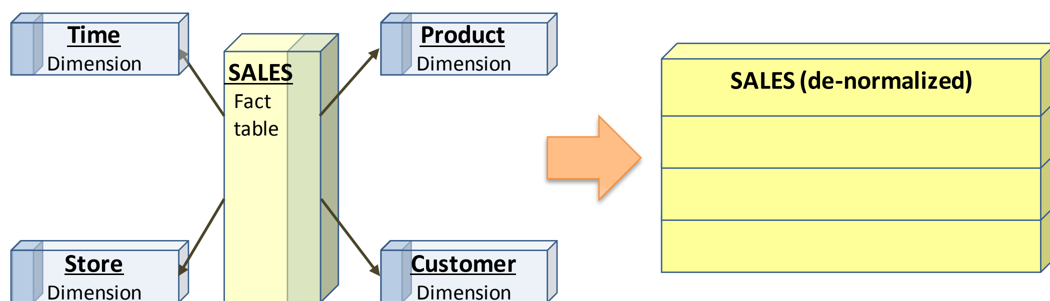


Fig. 3.2 – Star schema model vs ONE data model

ONE, since data is already joined (ONE ⇔ F ⋈ D), uses a simpler and predictable execution plan where $a_q = \pi\ (\sigma_q\ (\text{ ONE})\ )$. The join costs, which are unpredictable, are replaced by predictable IO cost of reading sequentially the ONE relation.

Query processing in ONE is scan-oriented. Data is mainly read sequentially from storage. Since the system is reading data sequentially, it is read at high rates. This change in IO pattern provides predictable query execution times, which can be determined as a function of the storage throughput, and also allows an enhanced data and processing sharing among concurrent queries (discussed in Section 3.4). Chapter 4 discusses ONE in detail.

3.2 Mechanisms for Providing Unlimited Scalability

Another source of unpredictability of the star schema is the data volume. An increase in data volume results in an unpredictable increase in query execution times, mainly because of the unpredictability of joining costs. Fig. 3.3, extracted from chapter 8, depicts the execution times for a set of TPC-H queries for a given data volume (5GB,

SF=5) and what happens when there a 2x increase in data volume (10GB, SF=10). The expected execution time, based on a linear proportion of the execution time with 5GB, and the actual execution time, are depicted respectively in red and blue. We can observe that the actual execution time is frequently larger than the expected execution time.

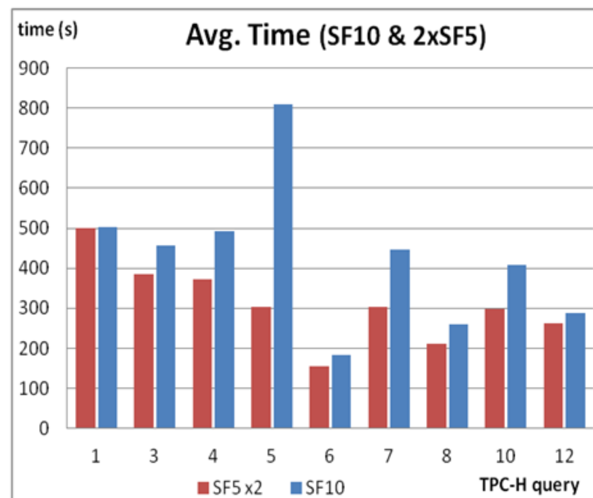


Fig. 3.3 – Execution time for a 2x increase in data volume

Larger DWs are frequently supported by parallel shared nothing organizations. As discussed in Chapter 2, fact tables can be partitioned into smaller fragments and allocated to nodes, while dimensions can be replicated or partitioned.

Scalability and performance speedup are constrained by the size of replicated dimensions and by the complex inter-node joins when dimensions are partitioned. This is especially relevant when managing large data volumes and multiple nodes. Since query processing involves operations with both facts and dimensions, adding nodes can limit the speedup.

To provide unlimited data scalability, we de-normalize completely and partition the de-normalized relation into data fragments distributed among a set of processing nodes for parallel processing. Fig. 3.4 illustrates ONE partitioned into multiple nodes in parallel infrastructures.

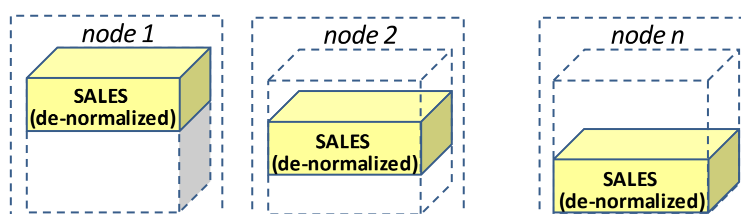


Fig. 3.4 – ONE fragments distributed among nodes

ONE delivers unlimited data scalability, since the whole data (fact and dimensions), and not just the fact table, is linearly partitioned among nodes (with η nodes, each will have $1/\eta$ of the data). The addition of more nodes triggers the data redistribution (re-balancing) among a wider set of processing nodes, without introducing any data overheads. For dependability or for query load balancing purposes, the architecture may consider the replication of some data fragments, particularly those that are more frequently used. Therefore, since the addition of more nodes to the processing infrastructure does not require additional data replication of dimensions, ONE provides massive data scalability.

By ensuring a linear distribution of the whole data, and not just the fact table, query execution time is improved proportionally to the data volume in each node. Moreover, since data in each node is already joined and thus query processing does not involve the execution of costly join algorithms, the speedup in each node is enhanced (almost) linearly as a function of the data volume that it has to process.

By de-normalizing the data we also decrease the nodes' requirements, in what concerns physical memory (needed for processing joins), and query processing tasks, since the join processing tasks that were repeatedly (over and over) processed are removed. The remaining filtering and aggregations tasks have minimum memory and processing requirements. Only group by aggregations and sorting have larger memory requirements. Queries are then executed using a simplified and predictable query processing approach, in a RISC-like manner [Patterson & Ditzel 1980]. We characterize the simplified query processing of the approach as a RISC-like capability by analogy to the simplified reduced instruction computing [Patterson & Ditzel 1980].

In summary, the ONE model presents several key advantages:

- No (in-memory or external) joins are required;
- Reduced memory requirements, mostly required for sorting and aggregation;
- Simpler processing model (RISC-like), composed by filter and aggregations
- Predictable, since it is based on predictable execution tasks
- Can be massively deployed over a parallel shared nothing infrastructure, with linear horizontal and vertical scalability. An increase in the number of

processing nodes linearly reduces the data load in each node, and therefore results in an almost linear drop in execution time;

- Data load and data processing is linearly divided among processing nodes
- Removal (or merging) of redundant keys (primary and foreign)

There are two issues that should be addressed in the context of ONE. It is scan-oriented and has larger storage space requirements. As ONE de-normalizes the star schema into a single relation, with dimension attributes being redundantly stored in the de-normalized relation, there is an increase in storage space requirements to hold the entire DW. However, this issue is minimized since ONE can be massively deployed over a parallel shared nothing infrastructure. Chapter 4 discusses ONE in detail.

3.3 Mechanisms for Providing Right-time

We define right-time as the ability to deliver query results in a timely manner, before they are required. The aim is not to provide the fastest answers, but to guarantee that the answers will be there (available) when expected and needed.

Definition 3.2

Let $t_{target}(q)$, denote the execution time target of query q, $\forall q \in Q$. We define that the system is able to provide timely results, or right-time execution, iff

$$\forall\, q \in Q,\ t_{exec}(q) \leq t_{target}(q)$$

The ability to provide right-time data analysis is gaining increasing importance, with more and more operational decisions being made using data analysis from the DW. The predictability of each of the query execution tasks is fundamental for providing right-time data analysis.

As discussed in Section 3.1, the proposed ONE data model is scan-oriented. The execution time is mostly influenced by the data volume that a node has to read and process. Therefore, for larger data volumes or tighter execution targets, a parallel infrastructure is needed. To meet specific right time targets, the number of processing nodes (p) and the data volume allocated to each node (v), which may differ according to the nodes' characteristics, have to be planned according to those targets.

$$\forall q \in Q,\ t_{exec}(q) = f\,(\pi\,(\sigma_q\,(\text{ ONE })\,,\, v,\, p,\, 1\,) \leq t_{target}(q\,)$$

The architecture includes a Timely Execution with Elastic Parallel Architecture (TEEPA) module, illustrated in Fig. 3.5, which provides right-time guarantees to aggregated queries. The figure shows TEEPA middleware, which communicates with a node service daemon (NS) running in each node to manage their availability and processing. TEEPA takes into consideration the query execution time targets, expressed at query or session level (details in Chapter 5), to adjust and rebalance the data volume and the processing infrastructure.

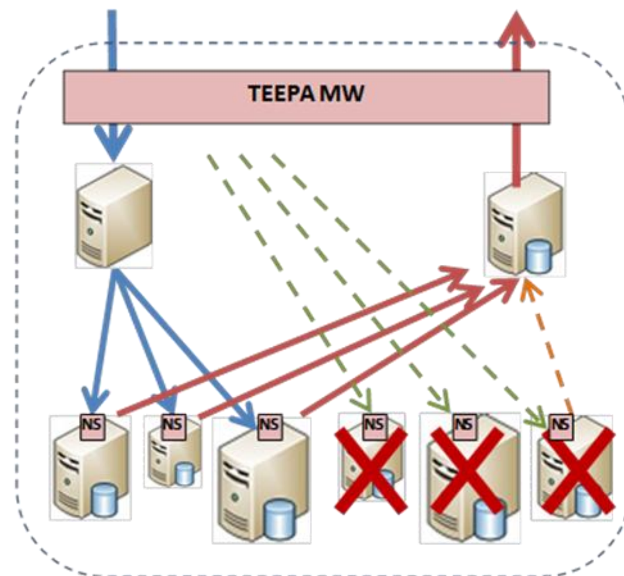


Fig. 3.5 – TEEPA framework

When the current deployment is unable to deliver the time targets, it adds more processing nodes and redistributes the data volumes among them. TEEPA continuously monitors the local query execution, the IO throughput and the data volume allocated to each processing node, to determine if the system is able to satisfy the user specified time targets.

When TEEPA determines that a query may miss the time target, it starts the node allocation and data rebalancing process. TEEPA begins by determining the maximum data volume that each node can process within the specified time target, and then it determines how many additional nodes are needed in order to process the whole data volume within the execution time target. After an additional node has been included in the parallel architecture, it sends to each node a set of data reallocation tasks, indicating the amount and the destination of the data to be rebalanced.

TEEPA was designed to handle heterogeneous nodes and thus it takes into account their IO capabilities when performing the necessary data rebalancing tasks. The

data volume allocated to each node is adjusted as a function of the whole data load (total number of tuples), the tuple size and the node' sequential scan throughput. Therefore, larger data volumes will be allocated to faster nodes. The node allocation (selection and integration of new nodes) and the data rebalancing tasks are continuously executed until the time target can be assured (as illustrated in Fig. 3.6). Chapter 5 discusses TEEPA in detail.

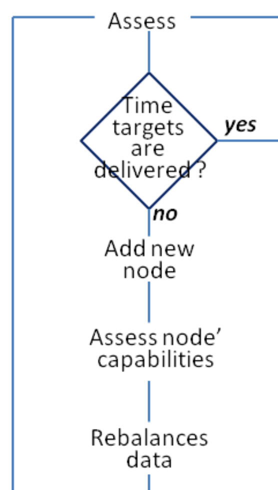


Fig. 3.6 – Data rebalancing process

3.4 Providing Freshness guarantees

Usually a DW is periodically refreshed (loaded) in batches, to reduce IO loading costs and also the costs related to refreshing the indexes and the pre-computation of aggregated data structures. However, there is an increasing demand for data analyses over near real-time data, with low latency and minimum freshness, which requires data to be loaded more frequently or loaded in a row-by-row fashion.

Main memory DBMS eliminate IO costs and thus can handle higher data loading frequencies. However, physical memory is limited in size and cannot typically hold the whole tables and structures.

To provide freshness guarantees, the proposed architecture combines a parallel ONE deployment with an in-memory star schema model holding recent data. The in-memory part (O_s) maintains the recently loaded data with minimum latency, thus allowing analysis to be carried out over almost real-time data. By using a star schema model in O_s, existing DW applications can be easily replaced and integrated with the architecture without the need to recreate the existing ETL tasks.

When the physical memory is exhausted, the data in O_s model is moved to O_d, from a star schema representation into the ONE model, as illustrated in Fig. 3.7. The data shift process is triggered when one of the following thresholds is not satisfied: the maximum memory threshold occupied by the data; and a maximum time interval between two flushes.

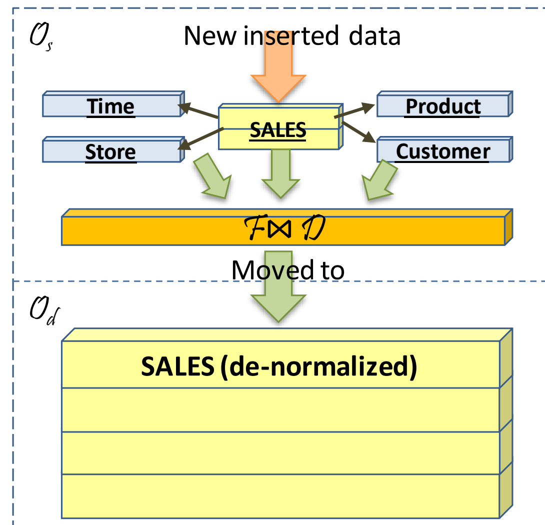


Fig. 3.7 – In-memory data buffering and ONE data flushing

Tuples from the fact table in O_s, after being de-normalized and loaded into O_d, are then deleted, to free memory for accommodating new fresh recent data.

Queries may require data that is stored in O_s (the in-memory part), in O_d (the de-normalized part) or in both parts. Section 4.5 details how the logical to physical layer manages data and query processing with this two schema models. The architecture includes a logical to physical layer that manages data and processing consistency between models, including the necessary query rewriting to process the data stored in each part, and merging the partial results.

From the user perspective and data presentation, the architecture offers a logical star schema model view of the data, in order to provide easy integration with existing applications and because the model has advantages in what concerns user understanding and usability.

3.5 Handling massive concurrent queries (query workload)

In the previous sections, we focused in providing massive data scalability and right-time guarantees for individual queries. However, modern DWs also suffer from

query load scalability limitations, as more and more queries (in particular ad-hoc) are being concurrently (simultaneously) executed. Larger parallel infrastructures can reduce this limitation, but its scalability is constrained by the query-at-time execution model of custom RDBMS, where each query is individually processed, competes for the existing resources (IO, CPU, memory,…) and access the common base data, without data and processing sharing considerations.

We propose SPIN, a data and processing sharing model that delivers predictable execution times for concurrent queries without the memory and scalability limitations of existing approaches. SPIN views a ONE relation, as a logical circular relation, i.e. a relation that is constantly scanned in a circular fashion. When it reaches the end, it continues scanning from the beginning, while at least one query is being executed. Data is read from storage and placed into an in–memory pipeline to be shared by all running concurrent queries, as illustrated in Fig. 3.8. Each query q, $q \in Q$, processes all tuples of relation ONE, but the scanning and the query processing does not start from the same first physical row. As the relation is read in a circular fashion, the first row processed by a query (first logical row) is the one that is cached in memory. The remaining tuples are processed as they are being read from storage until the first logical row is reached.

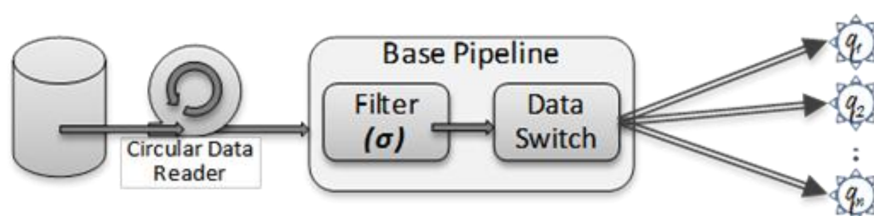


Fig. 3.8 – SPIN Data processing model

A Data Reader is continuously loading tuples from the ONE relation, and putting them in an in-memory data pipeline. Each query registers itself as a consumer of the data pipeline, following a publish-subscribe model, and starts processing the tuples as they pass through the pipeline. It registers the starting point (Fig. 3.9 depicts the starting point of queries q_1, q_2 and q_3), representing the first logical row (position in the circular relation). When a full loop is completed, it stops the query processing, de-registers the query as a consumer of the data pipeline and sends the query results to the user. Thus, IO reading cost is constant and shared between running queries. Therefore, the submission of additional queries does not introduce additional IO costs and joins operations. The execution time is influenced by the number of concurrent queries that

are currently running and also by complexity of the query constraints (filtering) and the cost of aggregations.

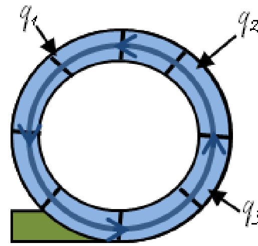


Fig. 3.9 – SPIN concurrent query processing model

Since the data is shared among processing queries, the workload scalability of SPIN is not constrained at IO level. To provide massive workload scalability it also has to share processing among queries. To achieve that goal, SPIN employs two different approaches: orchestrate query predicates in common logical branches, creating a workload processing tree and the reuse and merging of partial results from different branches.

The query predicates of a newly submitted query are analyzed to determine if the current workload has already created a logical branch with common predicates. If one exists, the query is registered as a consumer of that logical branch, and the corresponding query predicates are removed. Otherwise, if there exists no logical branch that meets the query predicates, it is registered as a new logical branch of the base data pipeline. This enhances processing sharing, and reduces the number of filtering conditions. SPIN has a branch optimizer that is continuously adjusting the number and order of the existing branches, and reorganizing them as required. Fig. 3.10 illustrates an example of a workload processing tree, composed with several branches that were built according to the query predicates that are being executed. The figure illustrates the WPTree created when queries q_1 (SUM($\sigma_{p=a}$(sales))), q_2 (SUM($\sigma_{y=2000}$ (sales))) and q_3 (SUM($\sigma_{y=2000 \wedge p=a}$ (sales))) are submitted. Branches are created for shared predicates, e.g. $\sigma_{y=2000}$ and orchestrated into a WPtree.

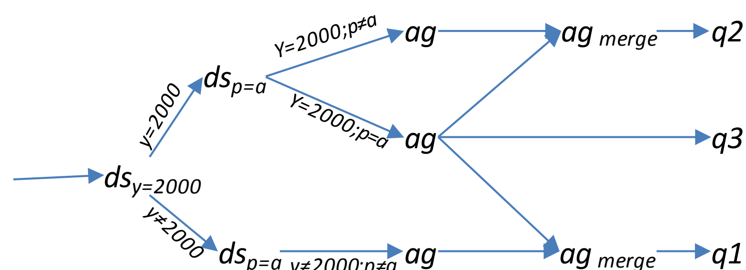


Fig. 3.10 – Branch processing

High accurate estimations of the query execution time can be calculated as function of the computational costs of each of the branches that belong to the query path (set of sequentially connected branches from the base pipeline). Therefore, predictable execution times can be given for massive workload scalability.

Whenever possible, a query can merge and combine results that are being processed by other branches, thus reducing the data volume that query branches have to filter and to process.

Tighter right-time guarantees can be provided by extending the parallel infrastructure, and redistributing the data volume amongst processing nodes, but also by redistributing queries, query processing and data branches between nodes that have replicated fragments. This is achieved by using two distinct approaches, a parallel fine-tuned fragment level processing, named CARROUSEL, and an early-end query processing mechanism.

CARROUSEL is a flexible fragment processor that uses idle nodes, or nodes that are running queries with looser execution targets, to process some of the fragments required by other queries that have tighter execution targets, on behalf of the fragment node's owner. By reducing the data volume to be processed by a node, we can provide faster execution times. The query processing costs can also be reduced by redistributing the processing of some logical data branches amongst nodes with replicated fragments.

The query execution ends when all tuples of the data fragments have been processed and the circular logical loop is completed. As the system is continuously spinning, reading and processing over and over the same data, it collects insightful information regarding the data that is stored in each data fragment. For some logical data branches, this information can be relevant to reduce memory and computational usage by using a postponed start (delaying the query execution until the first relevant fragment is loaded) and an early-end approach (detaching the query pipeline as soon as all the fragments that are relevant for a query have been processed), discussed in detail in chapter 7.

This information is also useful when the architecture needs to perform a data rebalancing process, in order to cluster the rebalanced data into new data fragments

according to the predicates of logical branches. Chapter 6 and 7 discusses SPIN in detail.

3.6 Performance and auxiliary data structures

Besides the main concepts presented above, the architecture also encompasses a set of auxiliary structures designed for performance and usability purposes. It exhaustively uses a set of in-memory bitmap-like indexes, named *bitsets*, which are built on-the-fly with the results of branch level processing (evaluation of predicates). Afterwards, subsequent evaluations of these predicates can be replaced with faster lookup operations.

This provides faster query results, since the evaluation of some filter conditions are substituted by bitmap positional lookups. These *bitsets* are also built at fragment-level to improve the rebalancing of data fragments.

3.7 Limitations of the proposed mechanisms

The approaches discussed in this thesis are based on de-normalization of the star schema. De-normalization is beneficial when the schema contains large dimensions and users require predictable and timely query execution.

De-normalization has some potential disadvantages. These include higher storage requirements and more IO dependent processing. This means that full de-normalization should not be used when all dimensions are small, because performance would be worse than using an ordinary star schema. To minimize this problem, we also considered the possibility to de-normalize the schema partially, according to the dimension sizes and whether they can be maintained in memory or not.

Parallelism can have an important role to reduce problems raised by the data volume, namely, higher storage requirements and the excessive time taken to read all that data. That costly task will be divided by many nodes, since the data is fully partitioned among the nodes of the processing infrastructure.

Modifying the number of nodes in TEEPA incurs in relevant IO, temporary space and network communication costs, due to data rebalancing needs, in order to

adjust the data volume and data layout according to the specific characteristic of each node.

By sharing data scans and processing of several queries, SPIN reduces the IO dependency in a multi-query environment. The major performance limitation of SPIN then becomes the CPU (CPU-bound). To reduce the amount of processing we also proposed some of the extensions to SPIN, so that it becomes extremely scalable even to a huge number of simultaneous queries.

3.8 Chapter Summary

This chapter described the main concepts behind the mechanisms of the proposed scalable architecture, to provide timely results for scalable data volumes, concurrent queries and near-real time requirements, and introduced the concept of timely data analysis (right-time execution). Fig. 3.11 depicts a general overview of the proposed mechanisms and how they integrate.

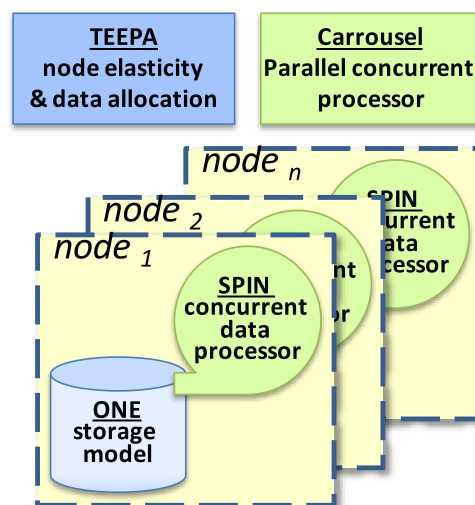


Fig. 3.11 – A high level integration view of the proposed mechanisms

The ONE data model can easily be used with any DBMS engine, but in order to be a seamless alternative to existing star schema models and for integration with existing client applications, it would require the use of a logical to physical rewriter module of TEEPA. SPIN is a processing engine that shares data and processing to allow the execution of a large number of concurrent queries, but it can also be used as an accelerator of existing DW infrastructures. The following chapters present a detailed discussion of the mechanisms outlined in this chapter.

Chapter 4

A Scalable and Predictable Data Model

For decades Data Warehouses (DW) have been deployed in relational database systems (RDBMS) using the *de facto* star schema model. Several factors contributed for its popularity and massive deployment, such as: ease of comprehension by both managers, power users and IT managers; the use of existing relational infrastructures (all the data is stored in relational tables) with minimum management and learning costs. This data organization, into fact and dimension tables, provides a performance-space tradeoff. However, today's increase in both data volume and query workload stresses those systems, limits performance and cannot take advantage of the advances in both storage and data processing capabilities.

In this chapter we propose ONE, a de-normalized data model, which trades storage space for predictable execution times to massive scalable data volumes. The chapter is organized as follows: Section 4.1 discusses the unpredictability factors that constrain the ability to deliver predictable execution times and handling larger data volumes; Section 4.2 presents the ONE processing model which combines the predictability and scalability characteristics of the de-normalized model, with the ease and user understanding of the star schema model; the ability of ONE to handle scalable data volumes and freshness requirements is discussed in Sections 4.3 and 4.4, respectively; Section 4.5 shows how queries are processed with the proposed approach; Section 4.6 discusses the changes in existing ETL tasks and how data can be loaded into ONE; Section 4.7 analyses the impact of the ONE model in what concerns storage requirements, Section 4.8 presents partial de-normalization approaches to reduce storage space requirements and Section 4.9 presents a summary of the chapter.

4.1 Unpredictability factors

The increasing data volume is stressing the DBMS engines in delivering predictable execution times, particularly because of the execution costs related to joining large relations that cannot fit in memory. The IO costs related to joins overkill the ability to provide predictable performance and scalable data volumes. To provide predictable execution times for scalable data volumes we first identify the unpredictability factors and how to overcome them.

The query execution time over a star schema is highly influenced by a set of factors, such as query selectivity, number and size of the relations that have to be joined, the join algorithms, the available memory for joining relations, and the additional IO operations (random and sequential) when the available memory is insufficient to process in-memory joins. The query optimizer has to choose the most appropriate execution plan and fine-tune the alternative execution plans with hardware characteristics, taking into account aspects such as the available memory for hashing and sorting. Fig. 4.1 depicts a possible star-query execution plan, which may change according to the selectivity and the required dimensions (not all may be needed).

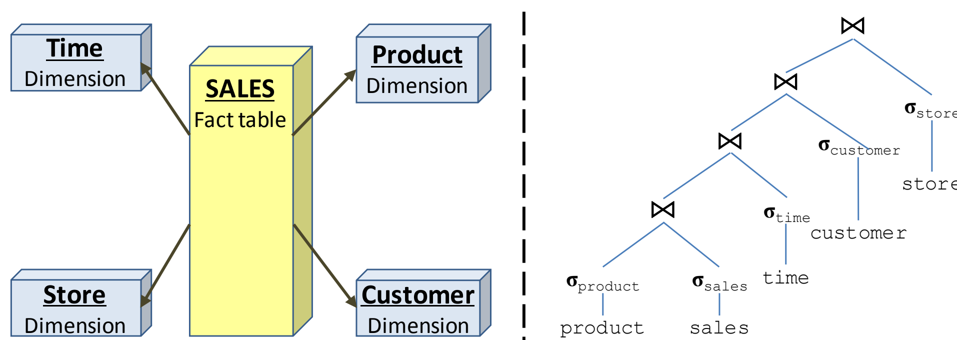


Fig. 4.1 – A typical star schema and an example of query execution plan

The query execution time involves the cost of joining all the required relations (fact and dimensions). For large relations this results in costly tasks and therefore in poor query execution times. Execution time is hard to predict, since it is highly influenced by each relation's size, selectivity and number of involved relations. The influence of these unpredictability factors is higher as the data volume increases.

Dimensions frequently occupy a small percentage of the overall DW storage space. However, with time, dimensions tend to be not so small, with new dimension characteristics (attributes) being added (e.g. add order validity date to table orders or a

new type of product characteristics), or with a larger number of tuples (e.g. for dealing with slowing changing dimensions where additional tuples are added for storing changes in dimension attributes) or both. The change in the size of dimensions also influences the cost of the execution plan.

Fig. 4.2, extracted from chapter 8, depicts the execution time of 9 queries of the TPC-H benchmark (1..9), with different query execution complexities. The figure depicts results for two distinct scale factors: SF1 (1GB) and SF10 (10GB, represents 10x times the data volumes of SF1). The primary y-axis shows the execution time for SF1, and the secondary y-axis shows the execution time for SF10. The scale of the secondary y-axis is 10 times that of the primary y-axis. From the figure we observe that for most queries the execution time with SF10 is more than 10x the execution time with SF1. Besides, different queries have quite different execution times. Therefore, to provide predictable execution time, we focus our research in an alternative data organization, with less unpredictable factors.

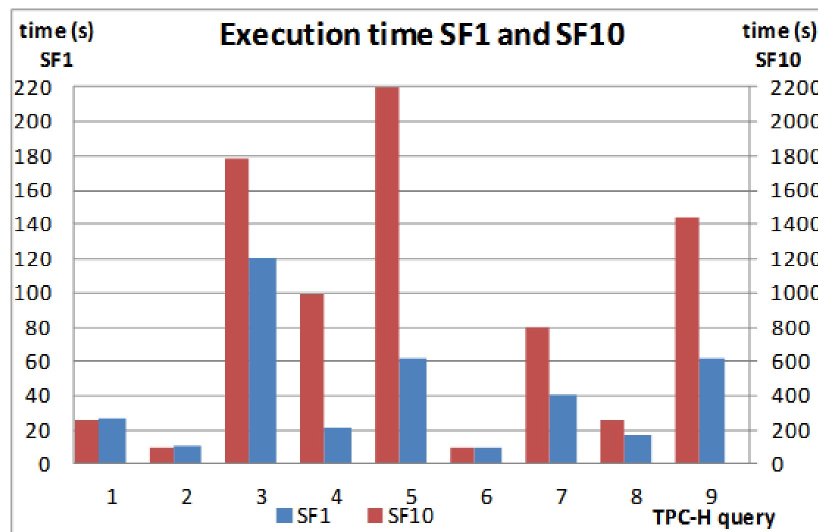


Fig. 4.2 – Execution time of different TPC-H queries

These unpredictability factors and joining costs arise from the star schema data model.

4.2 The ONE Data Model

To overcome such unpredictability factors and to provide predictable execution times, we propose to de-normalize the whole star schema model to be physically stored in a single (ONE) relation containing all the attributes from both the fact table and dimension tables without primary and foreign keys. Since no joins are required, we can

remove both the primary and foreign keys (key overhead). The cardinality of ONE, |ONE| is the same as the cardinality of the fact table, |sales| in the example. Fig. 4.3 depicts the star schema and the corresponding representation using the ONE data model.

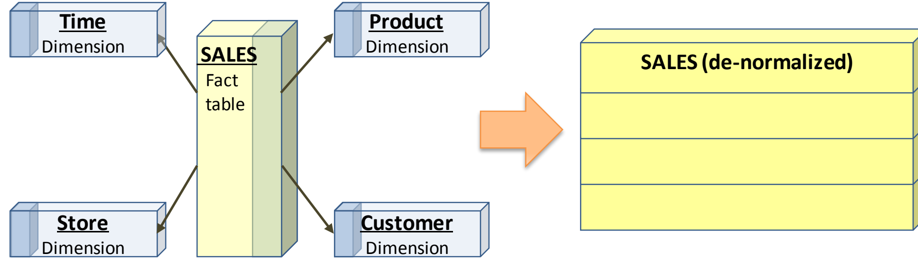


Fig. 4.3 – The star schema and the corresponding ONE data model

This de-normalization increases the overall space necessary for storing all the data, since data from dimension tables is redundantly sorted in ONE relation. For instance, in the TPC-H benchmark, each tuple of table CUSTOMERS is, on average, inserted (repeated) 40 times. This redundancy requires extra storage space for storing all the de-normalized data, and consequently may cause performance issues, since it now is more IO dependent. The ONE data model organization can be easily implemented using existing DBMS engines.

Definition 4.1

A star schema model S, where F denote the fact relation and $D=\{d_1,d_2,\ldots, d_n\}$ *denotes the set of dimensions, with d ranging on D. ONE denotes the de-normalized data model equivalent to S, such that* $ONE = \pi (F \bowtie D)$. *Let V be the data volume, C the number of queries that are currently being executed and P the processing infrastructure. The query execution time of q,* $t_{exec}(q)$, *is*

$$t_{exec}(q) = f(a_q, V, C, P),$$

where $a_q = \pi(\sigma_q(ONE))$, *represents the relational algebra representation of the query q (for simplicity of the formula we did not include aggregation, which can be added).*

With this data model, submitted queries can be processed through a more simplified query execution plan, since all the necessary join operations are removed and filter predicates are applied to the de-normalized relation. The execution plans are simpler, based only on predictable processing tasks over a known number of tuples, where the unpredictability factors related to joining the relations, namely the order of

joins, the join algorithms and the available memory, are removed. Fig. 4.4 depicts a side by side comparison of a typical execution plan for both data models: the star schema and ONE.

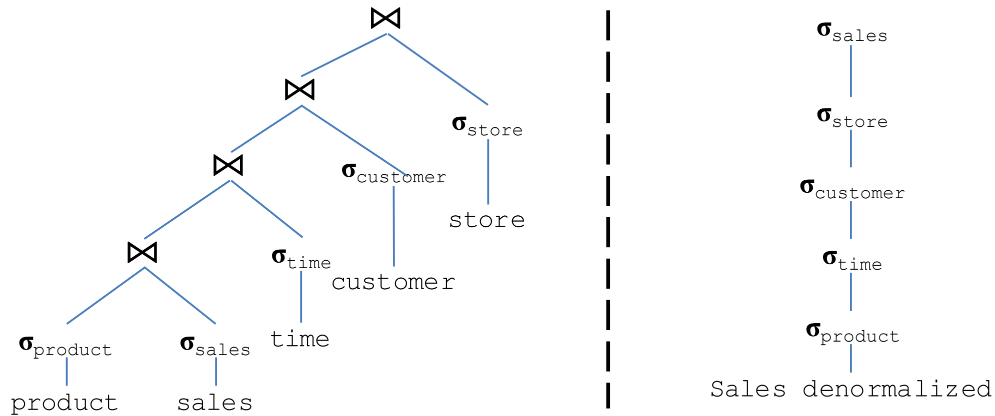


Fig. 4.4 – Query execution plan a) star schema b) ONE

Queries submitted against the ONE data model result in simpler execution plans, without joins, where selection tasks are placed according to their selectivity. In what concerns IO, query processing is then based on sequential scans, without slow random reads, yielding predictable execution times. Query execution time is dependent of system storage throughput and also the data volume. Fig. 4.5, extracted from chapter 8, depicts the execution time with the ONE data model for the queries shown above.

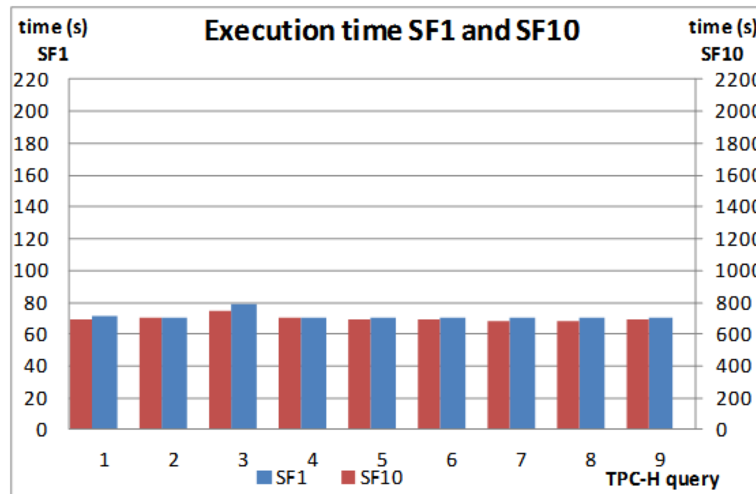


Fig. 4.5 – Execution time of different TPC-H queries with the ONE data model

As the data volume stored in DW systems is continuously growing in size, the ability to provide predictable execution time for scalable data volumes is important for DBA and IT managers to better estimate and determine the current limitations of existing hardware infrastructure and determine the requirements of the new infrastructure to handle a given data volume without even testing it.

4.3 Handling scalable data volumes

With a parallel infrastructure, the ONE de-normalized relation can be partitioned into data fragments and distributed among a set of processing nodes for parallel processing, thus providing performance gains. Fig. 4.6 illustrates the ONE data partitioning scheme in a parallel infrastructure.

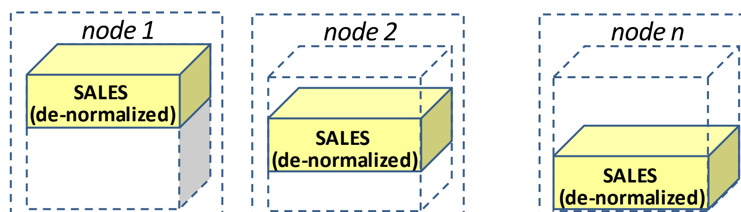


Fig. 4.6 – ONE fragments distribution among nodes

ONE scales-out almost linearly, since the whole data (fact and dimensions), and not just the fact table, can be linearly partitioned among nodes (with *n* nodes, each will have *1/n* of the ONE node). For dependability reasons or for query load balancing purposes, some data fragments may be replicated, particularly those that are more frequently used (fragment replication is discussed in more detail in chapter 5).

By ensuring a linear distribution of the whole data, and not just the fact table, query execution time is improved proportionally to the data volume in each node. Moreover, since data in each node is already joined and thus query processing does not involve the execution of costly join algorithms, the speedup in each node is enhanced (almost) linearly as a function of the data volume that it has to process. As discussed above, the performance of joins is highly influenced by factors such as the available memory (in-memory vs external joins) and the number of dimensions to join with the fact table.

As a result of using a de-normalized data model, we get a simplified query processing model with minimal memory requirements, only used for aggregations and sorting. Queries are then executed using a set of simple and predictable tasks.

4.4 Providing Freshness

Traditionally, DWs are periodically refreshed in batches, to reduce IO loading costs and costs related to the refreshing indexes and pre-computed aggregation data structures. However, there is an increasing demand for data analyses over near real-time

data, with low latency and minimum freshness, which requires data to be loaded more frequently or loaded in a row-by-row manner. Main memory DBMS eliminate IO costs and thus can handle more frequent data loads. However, physical memory is limited in size and cannot typically hold the whole tables and auxiliary structures.

Since the ONE data model has reduced memory requirements, we propose to combine a ONE data model (O_d) with an in-memory star schema model for holding recent data (O_s), as illustrated in Fig. 4.7.

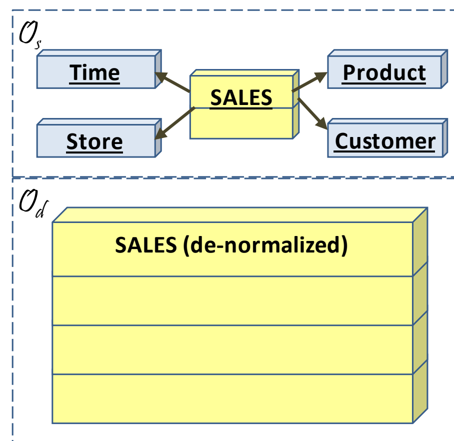


Fig. 4.7 – The hybrid O_s and O_d data model

O_s maintains the recently loaded data, which can be more frequently loaded with minimum loading costs, and thus allows the execution of most data analyses over fresh data with minimum latency. The use of O_s also allows an easier integration with existing DW applications without the need to recreate the existing ETL tasks. Data is loaded into the in-memory O_s and remains there for real-time processing while there is memory available, so that the most recent data is held in the star schema.

Since physical memory is limited and is rapidly exhausted, the data stored in O_s (in memory using the star schema model) has to be moved to O_d (in the ONE data model), as illustrated in Fig. 4.8. The data shift process is manually initiated by the DBA, or it can be triggered when one of the following thresholds is not satisfied: S_{memory} - the amount of physical memory occupied by the data stored in O_s (defaults to 30% of physical memory); $t_{denorminterval}$ - maximum time interval between two flushes (defaults to 10 min). Periodically, a watchdog daemon wakes up, every $t_{daemonCheck}$ (defaults to 10 sec), to check if the values defined by those parameters have been exceeded. When that occurs, a full O_s flush is done.

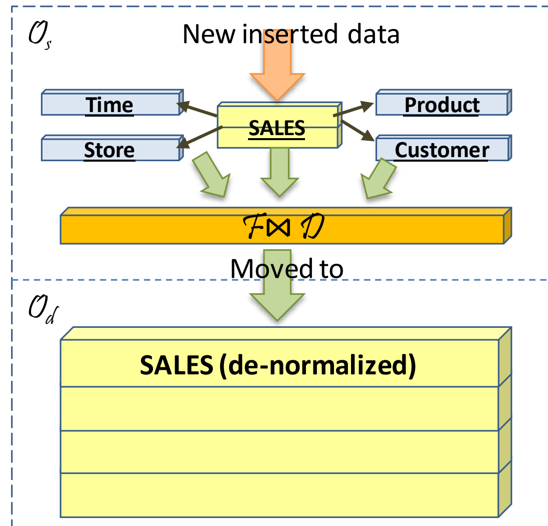


Fig. 4.8 – In-memory data buffering and ONE data flushing

Flushing is done by submitting a query q against O_s, which performs a natural join ($F \bowtie D$) between the fact table F and all dimensions D. The result is materialized into de-normalized O_d. Afterwards, every fact tuple t_f that has a corresponding tuple t_o in the de-normalized relation O_d, such that $\exists\ t_o \in O_d,\ t_f \in F,\ t_f \subseteq t_o$, is removed from the fact table F.

Tuples from the fact table in O_s, after being de-normalized and loaded into O_d, are deleted from O_s to free memory to accommodate new fresh data. In what concerns dimensions D, three approaches are available: left unchanged, remove all dimensions tuples, remove all dimension tuples t_d unreferenced by fact tuples t_f, or a combination of both, i.e. tuples t_d of a subset of dimensions D_r, $D_r \subset D$ are removed, while the remaining dimensions D_u ,$D_u \subset D$, $D_r \cap D_u = \emptyset$, are left unchanged. The former has the disadvantage that subsequent queries q, although F being smaller, experience poor performance since F has to be joined with relatively larger dimensions ($F \bowtie D$) in comparison with F. By removing unreferenced tuples from dimensions D, they will became smaller and consequently the join processing cost of $F \bowtie D$ will be significantly smaller.

4.5 ONE Query Processing

A query may require data that is stored in O_s, in O_d or in both parts. The architecture includes a logical to physical layer that manages data and processing consistency between models, including the necessary query rewriting for querying the data stored in each part, and merging of results. From the user perspective and data

presentation, the architecture offers a logical star schema model view of the data, in order to provide easy integration with existing applications and because the model has advantages in what concerns user understanding and usability.

Query processing has to take into account these two distinct data structures: the in-memory star schema structure buffer O_s $(S\ (F;\ D)\ )$ that temporarily holds the newly inserted data, and a de-normalized relation O_d holding the remaining data after it is de-normalized. The data is de-normalized opportunistically, based either on memory size limit or time intervals. O_s also represents a logical view of the star schema model, providing a seamless integration with existing applications.

Since the processing model is composed by two data structures, each query q is rewritten and decomposed into two distinct queries, q_s and q_d, and each executed against the respective part of the model $(O_{s,}\ O_d)$. The results are then combined with a merging query q_m to deliver the query result, as illustrated in Fig. 4.9.

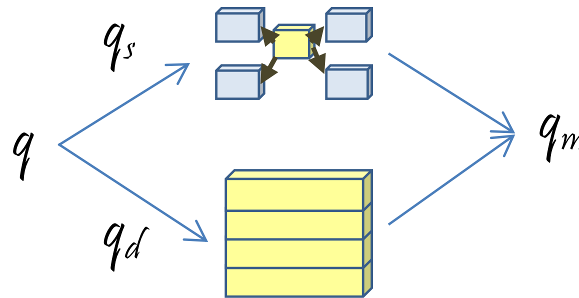


Fig. 4.9 – Decomposition of query *q*

An aggregation query $q \in Q$, which takes the form of

$q\text{: } G_1, G_2, \ldots, G_m, g_{f1\ (a1'), f2\ (a2'), \ldots, fk\ (ak')}\ (F \bowtie D), \text{ where } a_j', 1 \leq j \leq k, a_j' \in A$

where $G_1, G_2, \ldots, G_m$ *are group by attributes,* $g_{f1\ (a1'), f2\ (a2'), \ldots, fk\ (ak')}$ are aggregation functions, e.g. SUM, COUNT,…

is rewritten into 2 distinct sub-queries, q_s and q_d, one for each part of the ONE model $(O_{s,}\ O_d\ )$, and a merging query q_m , for merging intermediate results from both parts. q_m is only built when $O_s \neq \emptyset \wedge O_d \neq \emptyset$.

q_s is similar to q, except for the aggregations functions $f_k\ (a_k')$ which have to be rewritten as a set of functions $f_k‘(a_k')$ that take into consideration the need to merge partial aggregation results. For instance, an average aggregation $avg(a)$ has to be

rewritten as *sum*(*a*) and *count*(*a*), so that the merging query be able to compute the average as $(sum(a)_{Od} + sum\ (a)_{Os})/ (count\ (a)_{Od} + count(a)_{Os})$. The query selection predicates of q are left unchanged in qs. The resulting query qs is,

$q_s : G_1, G_2, \ldots, G_m, g_{f1\ '(a1'), f2\ '(a2'), \ldots, fk\ '(ak')} (F \bowtie D)$, where a_j', $1 \le j \le k$, $a_j' \in A$

Q_d is rewritten in the following manner: in what concerns aggregation functions, the rewriting process is similar to q_s. However, the join conditions are removed and the relations in the FROM clause are replaced with O_d.. The resulting query q_d is,

$q_d : G_1, G_2, \ldots, G_m, g_{f1\ '(a1'), f2\ '(a2'), \ldots, fk\ '(ak')} (O_d)$, where a_j', $1 \le j \le k$, $a_j' \in A$

The merge query q_m processes the partial results $(q_d \cup q_s)$ of q_d and q_s, to compute the result of query q.

$q_m : G_1, G_2, \ldots, G_m, g_{f1\ (a1''), f2\ (a2''), \ldots, fk\ (ak'')} (q_d \cup q_s)$, where a_j'', $1 \le j \le k$, are the intermediate aggregation results

4.6 Changes to the ETL process

ETL processes, after extracting and cleansing the data, usually have to perform the following steps:

- Step 1 - Load tuples into dimension tables
  For a new tuple, a new surrogate id is generated to be used as a dimension key before loading it into the dimension table. An updated tuple is handled according to the dimension SCD (slowly changing dimension) strategy. With a type 2 or type 4 strategy, data is inserted as new tuples, whilst with a type 1 or type 3 strategy the existing (already loaded) tuples have to be updated according with the new data values.
- Step 2 - Load tuples into the fact table
  For new data values (measures), a set of lookup key tasks are required to obtain the corresponding foreign key values, one for each dimension, before it can be loaded into the fact table.

Since we use two distinct data structures (O_s and O_d), new data can be loaded using two distinct approaches: loading into O_s, which uses a star schema model, or loading directly into O_d.

Loading the data into O_s offers better transparency and easy replacement of existing star schema implementations, since the existing ETL plans can be used without changes. As discussed in section 4.4, the data in O_s afterwards will be moved to O_d. However, until the data is flushed from O_s to O_d it cannot take advantage of the O_d representation.

For loading new data directly into O_d, a new ETL plan has to be built, which takes into consideration the data format of O_d, that includes a transformation step that performs a natural join of the new data before loading it into the de-normalized table. The plan does not require the typical steps related to the generation of dimension key values, and for each new fact tuple, the subsequent fetching of the corresponding dimension key values from all dimension tables. With a de-normalized O_d, the costs with the generation of key values, the subsequent key lookup, the enforcing of the dimensions SCD strategy and the rebuilding of key based indexes are avoided.

Regarding SCD, with the de-normalized data model O_d, we follow a insert only policy, rare updates are only admissible to correct erroneous values that were previously loaded. The data is loaded as a continuous time-snapshot historical log, i.e, each fact tuple t is joined with the current value of the dimension tables, that reside in O_s, before being loaded. Therefore the ETL plans for loading into O_d are simpler and be processed in parallel.

4.7 Storage size requirements

The star schema uses a normalized central fact table, which reduces the overall DW storage size, since it only stores a set of measures ($m_{measures}$), which are mainly numerical attributes (facts) and a set of foreign keys ($n_{foreignkeys}$) that are also numerical identifiers. The size of the fact table (ss_F) increases as a function of the number of tuples.

$$ss_F = ss(F) = |F| \times (length(n_{foreignkeys}) + length(m_{measures}))$$

The star schema model includes a set of primary and foreign keys, which are usually generated artificially (surrogate keys) and do not have operational meaning, to allow the join of fact with dimension tables. These keys are required and represent a storage space overhead of the star schema. In most scenarios, the number of foreign

keys represents a large percentage of the fact table attributes. For instance, a factless table, a fact table that doesn't have measures, is entirely composed with foreign key attributes.

Beside the overhead related with storing the foreign key attributes, we also have to account for other key related overheads related to key indexes, because most RDBMS engines automatically create indexes on unique and primary key attributes. Fig. 4.10, extracted from experimental section 8.2.1, which illustrates the storage size distribution of the TPC-H schema, shows that the storage size overhead can reach up to 25% of the relations' size.

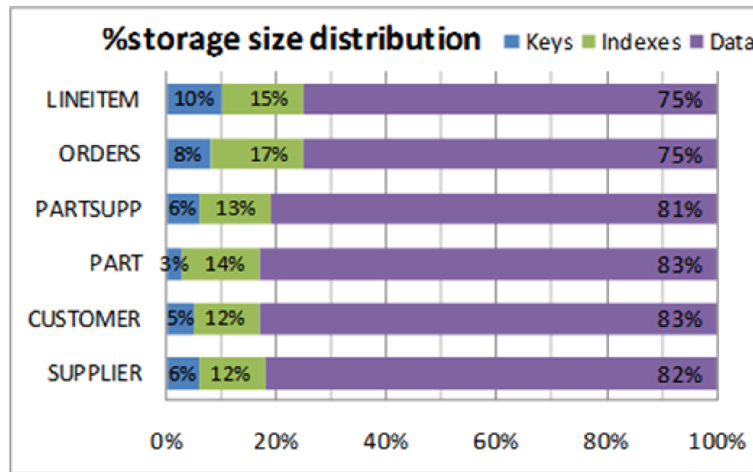


Fig. 4.10 – Data size distribution of the TPC-H schema

Considering ss_{DW} as the storage space requirements of a schema model, the total storage space occupied by a DW is obtained as the sum of the tables and indexes.

$$ss_{DW} = ss_{tables} + ss_{pk_{indexes}} + ss_{fk_{indexes}}$$

Where ss_{tables} is the size of the fact table, $ss\ (F)$, and related dimensions, $\sum ss(d_i), d_i \in D$. The size of each dimension is determined as

$$ss\ (d_i) = |d_i| \times (length\ (_{primarykey}) + length\ (mi_{attributes}))$$

The storage space required by the ONE data model is determined as

$$ss_{ONE} = ss_{Od} + 0 + 0$$

Where the storage size of the ONE data model is determined as

$$ss_{Od} = |F| \times (\ length\ (m_{measures}) + \sum length\ (mi_{attributes})\ )$$

We define φ_{ss} as the storage space increase ratio in comparison with the base DW star schema model

$$\varphi_{ss} = \frac{ss_{ONE} - ss_{dw}}{ss_{dw}}$$

Using as an example the TPC-H schema, shows the storage requirements of TPC-H, with and without indexes, and the equivalent de-normalized data model representation. The ONE data model represents a 4,47x increase in the storage size in comparison with the base tables, but when we also consider the space required by key related indexes, it is reduced to about 3,32x times.

Schema (SF1)	Size (MB)	φ_{ss}
base TPC-H	1.144,7 MB	4,47
base TPC-H + Indexes	1.448,4 MB	3,32
ONE	6.270,0 MB	

Table 4.1 – Storage space required by each schema organization

For quite some time, this increase in storage space was unacceptable since storage space was expensive, disks had limited capacity and with slow transfer rates. However, current disks are much faster, providing sequential transfer rates of hundreds of MB per second, at affordable prices (with prices below 0.05€/GB). With the ONE data model, a query has to read larger data volumes from storage, perform less equality predicates to join relations and are bound-free from memory requirements needed by join algorithms, such as sort-merge and hash joins, which may require the use of costly random IO operations as well.

4.8 Partial de-normalization

In some scenarios, where dimensions have large width but reduced cardinality (reduced number of tuples) and can be loaded entirely in memory, a partial de-normalization approach can be used to reduce the storage size requirements. This section presents two distinct approaches for partially de-normalizing the star schema.

4.8.1 Partial de-normalization of larger dimension tables

Dimensions of a star schema (D) have distinct characteristics. Some dimensions D_S ($D_S \subseteq D$) are small and can be efficiently joined using in-memory join algorithms,

but frequently there is a subset of dimensions D_L ($D_L \subseteq D$, $D_s \cap D_L = \emptyset$) that do not fit exclusively in memory and to which the join of ($F \bowtie D_L$) may need to store temporary results to disk, thus significantly degrading query performance. If D_s are small and can reside exclusively in memory, the de-normalization can be applied only to the larger dimensions D_L. In this case, O_d will be a de-normalized model of (F, D_L), which contains the attributes $A_o = A_F \cup A_{DL}$, without attributes from smaller dimensions A_{DS}. Fig. 4.11 illustrates a partial de-normalized schema, where all dimensions are de-normalized, except the STORE dimension, which is small and contains a reduced number of tuples.

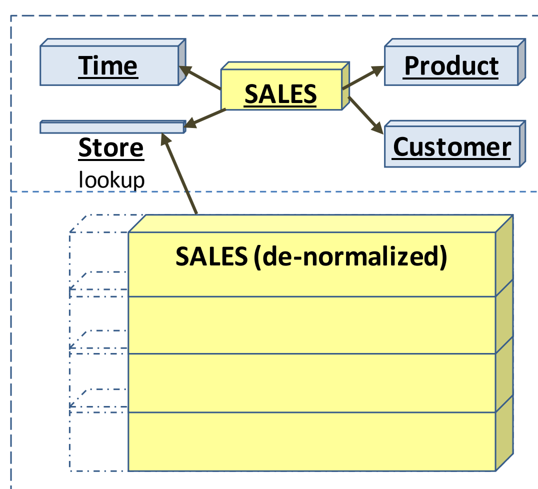


Fig. 4.11 – Partial de-normalized schema

With a partially de-normalized schema, large dimensions are stored in O_d, thus avoiding costly joins, and the remaining dimensions are joined using fast in-memory joins. Since a partial de-normalized O_d is narrower than the full de-normalized relation, it has less storage space requirements, may yield faster IO performance (O_d tuples per second), but has higher memory requirements, for holding D_s and carrying out joins.

To determine if it is beneficial to de-normalize two relations, we compare analytically the cost of reading and joining these relations, with the cost of reading the already de-normalized data. For joining relations we used the hybrid hash join, which, as discussed and evaluated in [DeWitt et al. 1984], is a join algorithm that delivers enhanced execution time.

Definition 4.1

Considering two relations R and S, let t_{join} (R,S) be the time needed to join R with S, and t_{ONE} (R,S) the time to read the equivalent de-normalized data ($R \bowtie S$). It is beneficial to use a de-normalized data model when $t_{join}(R,S) > t_{ONE}(R,S)$.

Hash Join algorithms use a hash function to partition relations *R* and *S* into hash partitions and are particularly efficient for joining large data sets [DeWitt et al. 1984]. The optimizer selects the smaller relation as the inner relation, used as the lookup driver relation, and builds a hash table in memory based on the join key. It then scans the larger table, probing the hash table to find the joined rows. This method is best used when the smaller table fits entirely in memory. The optimizer uses a hash join to join two tables if they are joined using an equijoin and a large amount of data needs to be joined together.

When the available memory is insufficient to hold the entire inner relation, the Hybrid Hash Join algorithm splits both relations into partitions, such that a hash table for a partition of the inner relation can fit entirely in memory. Corresponding partitions of the two input relations are then joined by probing the hash table with the tuples from the corresponding partition of the larger input relation. Partitions that cannot fit into memory have to be temporally written to disk, before being joined together.

Definition 4.2

Consider R as a dimension d, $d \in D$, and S the fact table F, and R is smaller than S. For relation R, consider that α_R is the number of tuples, ρ_R is the number of tuples that can fit in a block (or page) with size ς; β_R is the number of blocks (or pages) and ξ_R is the tuple size of R.

The cost of joining relations *R* with *S*, using a Hybrid Hash Join algorithm [DeWitt et al. 1984][Patel et al. 1994] can be computed as

$$
\begin{aligned}
\mathrm{HHJ}(R,S) = {} & (\alpha_R + \alpha_S) \times I_{hash} \\
& + (\alpha_R \times \xi_R + \alpha_S \times \xi_S) \times (1 - q) \times I_{copy} \\
& + 2 \times (\beta_R + \beta_S) \times (1 - q) \times IO \\
& + (\alpha_R + \alpha_S) \times (1 - q) \times I_{hash} \\
& + \alpha_R \times \xi_R \times I_{copy} \\
& + \alpha_S \times I_{probe} \times F_{hash}
\end{aligned}
$$

Where I_{hash}, I_{copy} and I_{probe} represent the number of instructions needed to perform the corresponding operation.

Considering $q = \beta_{R_0}/\beta_R$, and β_{Ro} as the size of the first partition that can reside in memory, which doesn't need to be written to disk, and IO the cost of retrieving a page from disk. The cost of processing a join between relations R and S can be determined as,

$$t_{join} = (\beta_R + \beta_S) \times IO + HHJ(R,S)$$

Using ONE data model, the cost for executing the same query, without considering filters and computations, can be determined as

$$t_{ONE} = \frac{\alpha_S}{\psi(\xi_S + \xi_R)} \times \varsigma \times \mathrm{IO}$$

with $\psi(\xi_R) = \left\lfloor \frac{\varsigma}{\xi_R} \right\rfloor$, but assuming that $\left\lfloor \frac{\varsigma}{\xi_R} \right\rfloor \cong \frac{\varsigma}{\xi_R}$, then

$$t_{ONE} = \frac{\alpha_S \times (\xi_S + \xi_R)}{\varsigma} \times \mathrm{IO}$$

When the cost of joining a dimension *d* with the fact table *F*, is greater than $\mathrm{IO} \times \beta_d \times \left(\frac{\alpha_F}{\alpha_d} - 1\right)$, then the F⋈d should be de-normalized. The *partial_de-normalization* algorithm shown below evaluates if it is beneficial to de-normalize each dimension, *d:d*∈*D*, with *d* ranging in D, from the largest to smallest one.

Algorithm: *partial_de-normalization*
De-normalization of largest dimensions

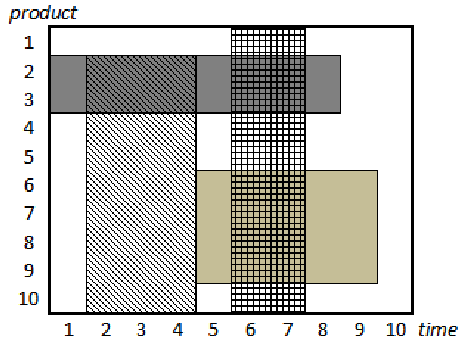
Input: *F* – the fact relation
Input: *D* – set of dimensions sorted according to its size (from largest to smallest)
Output: O_d – the de-normalized schema
Output: *D'* – set of dimensions not de-normalized, $D' \subseteq D$, $|D'| \geq 0$

1 $O_d \leftarrow F$
2 **foreach** *d* in *D* **do**
3 | **if** $t_{ONE}(d, O_d) > t_{join}(d, O_d)$ **then**
4 | | $O_d \leftarrow O_d \bowtie d$
5 | | $D' \leftarrow D' \cup \{d\}$;

4.8.2 Workload-driven de-normalization

Workload-driven de-normalization takes into consideration the query pattern workload *Q* submitted against O_s to determine frequently used query predicates. Considering *A* as the set of all the star schema attributes, where A_D are dimension

attributes and A_F are fact attributes, such that $A = A_D \cup A_F$, the goal is to determine the set of attributes A_p , such that A_p={ $a_1,\ldots, a_k$ }, $a_k \in A$, that are frequently used by the query workload. For attributes A_p, it determines the set of most frequent values. Top k subsets A_p are selected for de-normalization, with each subset A_p representing distinct data ranges. For example, Fig. 4.12 illustrates a set of A_p for a given workload with k=4 and considering a 2-dimensional data space (product; time).



$$A_1 =\{(\,prod \geq 2 \vee prod \leq 3\,) \wedge\ time \leq 8\ \}$$
$$A_2 =\{(\,prod \geq 6 \vee prod \leq 9) \wedge (\,time \geq 5 \vee time \leq 9\,)\}$$
$$A_3 =\{(\,time \geq 6 \vee time \leq 7\,)\}$$
$$A_4 =\{(\,prod \geq 2 \vee prod \leq 10\,) \wedge (\,time \geq 2 \vee time \leq 4\,)\}$$

Fig. 4.12 – Example of a set of A_p with k=4 in a 2-dimensional data space

The de-normalization is performed when a new query q, which has predicates P that match a subset of A_p, is submitted. Its execution plan is altered in order to push forward the predicates P just before join of ($F \bowtie D$). Therefore, the intermediate result of $F \bowtie D$, represents a fragment (a subset) f_p, with predicates p of the overall de-normalized data. For instance, a query q such that $\sigma_q \cap A_4 = A_4$, then the intermediate result of $\sigma_{A4}(F \bowtie D)$ is moved to O_d as a new data fragment f_p.

A metadata repository M maintains information regarding which subsets of the overall data space A_p have been de-normalized, and where each fragment f_p is stored. After a f_p is made available for processing and until the corresponding data tuples t_f are not removed from the fact table F, subsequent sub-queries q_s that are submitted against O_s are rewritten to exclude the query predicates of the f_p fragment.

$$\sigma_{p_q} \rightarrow \sigma_{p_q \wedge f_p}$$

As a consequence, the data volume of both F and D is reduced, and thus improves the join cost.

The query planner uses the information stored in the meta-data repository M, regarding the fragment predicates, where it is located, and in which data model, in order to rewrite the queries accordingly.

4.9 Chapter Summary

This chapter described ONE, a de-normalized data model, which trades storage size for predictable execution times of massive scalable data volumes and without the unpredictability issues of the star schema model. We described the ONE processing model which combines the predictability and scalability characteristics of the de-normalized data model, with the ease and user understanding of the star schema model, and described how data freshness can be achieved.

We discussed the impact of the model in storage size requirements, in the existing ETL tasks, and presented alternative methods for loading new fresh data. We also described methods for processing partial de-normalization of the schema.

ONE data model offers predictable execution time, and scales linearly with the volume of data.

Chapter 5

Providing Right-time with a Elastic Parallel Architecture

The query execution time is constrained, among other factors, by the volume of data that has to be processed. To handle huge amounts of data with acceptable response times, usually we have to use parallel shared-nothing architectures. They yield good performance and scalability capabilities, by distributing data among nodes to maximize the local computation of partial results. With a star schema model, while the fact table is partitioned into smaller partitions and allocated to nodes, dimensions can be replicated into each node. The replication of dimensions, particularly large dimensions, constrains the system scalability, and therefore some sort of execution time guarantees can only be provided by over-dimensioning the processing infrastructure. This issue is even more relevant when the processing infrastructure is composed by heterogeneous nodes.

To provide execution time guarantees, we split the ONE data model into data partitions and distribute them over a set of processing nodes. Since the ONE data model provides predictable performance, the data volume allocated to each node is proportional to the node ability to process it in a timely manner. We also propose TEEPA (Timely Execution with an Elastic Parallel Architecture) that uses an elastic set of heterogeneous nodes to provide right-time execution guarantees. We describe the mechanisms that verify when the current deployment is unable to provide timely execution times, that determine how many nodes, and which nodes, have to be integrated into the parallel deployment, and also the data volume that has to be redistributed amongst nodes.

5.1 Introduction

In a parallel shared nothing deployment, each node computes partial results based on the data that is stored locally. A merger node waits for these partial results, and then merges them to compute the query result, before sending it to the user. Therefore, each submitted query q has to be rewritten into a set of sub-queries that computes the partial results (q_p) in each node, and a merging query that merges the partial results (q_{pR}) to gather the query result (q_R).

Definition 5.1

Consider a parallel deployment P with η nodes, $P=\{node_j\}_{j=1,\ldots,\eta}$. For a query q, $q \in Q$, let t_{exec} be the query execution time, t_{rw} be the time needed to rewrite and build the partial queries, t_{tpq} the time to transfer the partial query to each node, t_{n_j} the local execution time in a node j; t_{tpr} the time required by a node to transfer its partial results to the merger node; t_m the time required by the merger node to receive and merge the partial results and to compute the final result and t_s the time required to send the final results. The query execution time $t_{exec}(q)$ is computed as

$$t_{exec}(q) = t_{rw}(q) + \sum_{j=1}^{\eta} t_{tpq_j}(q_p) + max\left(\left\{t_{n_j}(q_p) + t_{tpr_j}(q_{pR})\right\}_{j=1}^{\eta}\right) + t_{m_j}\left(\sum_{j=1}^{\eta} q_{pR}\right) + t_s(q_R) \quad (1)$$

Assuming that t_{rw}, t_{tpq} and t_s are negligible then t_{exec} can be estimated as

$$t_{exec}(q) = max\left(\left\{t_{n_j}(q_p) + t_{tpr_j}(q_{pR})\right\}_{j=1}^{\eta}\right) + t_{m_j}\left(\sum_{j=1}^{\eta} q_{pR}\right) \quad (2)$$

The overall query execution time $t_{exec}(q)$ is mostly influenced by the local query execution of the slowest node $t_{maxlocalexec}$, determined as $\max\left(\left\{t_{n_j}(q_p)\right\}_{j=1}^{\eta}\right)$, the number of nodes η, the partial results' size and the cost of sending them to the merger node.

The local query execution time t_n can be improved by increasing the number of processing nodes η of the parallel infrastructure, and thus reducing the amount of data that each node has to process. However, to allow independent local query processing of star schema DW, only the fact table is split among nodes whereas dimensions are fully replicated. As dimensions are replicated, their relative weight in storage space in each

node, and consequently in the local query processing time, increases with the number of nodes to a level where adding more nodes represents a minimal local performance improvement. Moreover, the costs related to exchanging and merging partial results increases with the number of nodes, gaining an increasing weight in the overall query execution time t_{exec}. Equi-partitioning may help in this matter, by partitioning both the fact table and some large dimension on a common attribute, usually the dimension primary key, and allocating related partitions into the same node. However, since business data inherently do not follow a random distribution, and data is skewed, the used equi-partitioning may introduce another limitation to scalability, with some nodes storing more data than others.

Definition 5.2

For a query q, $q \in Q$, let t_{exec} (q, P) be the query execution time over a parallel infrastructure P composed with |P| nodes, and t_{target} (q) be the query execution target. We define that the parallel infrastructure P can meet the query q time target iff $t_{exec}(q, P) \leq t_{target}(q)$. $t_{exec}(q, P)$ is constrained by $t_{maxlocalexec}$ the execution time of the slowest node.

5.2 Speeding up the ONE data model – ONE-P

The ONE data model has minimal processing requirements and delivers predictable execution time, as it is highly influenced by the storage IO capacity (making it IO dependent), but it does not fully explore the individual computational and memory capabilities of existing heterogeneous nodes. ONE-P extends the ONE data model by partitioning it horizontally and distributing the partitions among processing nodes. Since ONE-P does not require complex join algorithms (data is already joined), it has reduced memory requirements, and it can be employed with a wide set of non-dedicated processing nodes.

Definition 5.3

Consider a data volume V to be deployed over P. Let v_i be the data volume allocated to node i, $v_i=f(V)$, $V=\sum v_i$, and let $t_n(i, v_i)$ be the time required by node i to read and process the data volume v_i. Let $t_{maxlocalexec}$ be the maximum local execution time of processing nodes, $\forall i \in [1;\eta]$: $t_{maxlocalexec} \geq t_n(i, v_i)$. The goal is to determine the number nodes η that are needed and the data volume v_i

allocated to each node i, in order to guarantee that $\forall i \in [1;\eta]$, $t_n\ (i, v_i) \leq t_{maxlocalexec}$.

ONE-P horizontally partitions the ONE data model into smaller and manageable partitions, which are distributed among processing nodes. ONE-P does not require additional storage space since the ONE relation is horizontally partitioned without overheads between processing nodes. Fig. 5.1 illustrates a parallel deployment of ONE-P, with each node containing a data fragment.

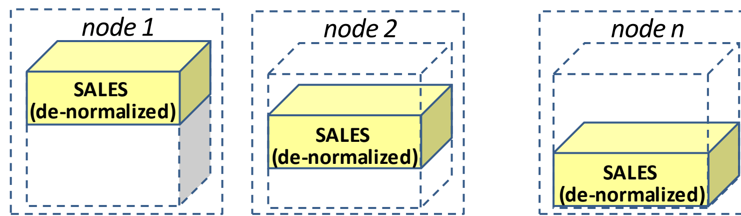


Fig. 5.1 – ONE fragments distribution among nodes

Since ONE is a single relation, it can be freely partitioned without constraints and the scalability limitations introduced by the replication of dimensions of the star schema. ONE-P scales-out almost linearly, since the whole data (fact and dimensions), and not just the fact table, can be linearly partitioned among nodes (with η nodes, each node will have $1/\eta$ of the ONE-P). Fig. 5.2, extracted from the experimental chapter, depicts the speedup comparison between the parallel deployment of a TPCH-P schema and the equivalent ONE-P data organization, over a variable number of processing nodes.

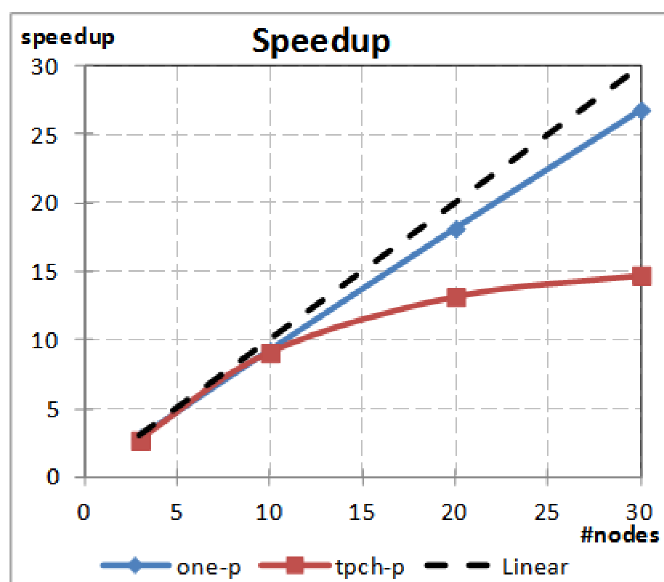


Fig. 5.2 – Speedup comparison between TPCH-P and ONE-P

By ensuring a linear distribution of the whole data, and not just the fact table, query execution time is improved proportionally to the data volume in each node. In contrast with the star schema, with ONE-P there is no data overhead when we vary the number of processing nodes. The overall data size required by ONE-P remains constant, while the overall size of a parallel deployment of a star schema DW increases as a function of the number of nodes.

With a parallel deployment composed with homogeneous nodes, and since the overall storage size is constant, regardless of the number of nodes, the storage size in each node decreases linearly as a function of the number of nodes. The data volume allocated to a node and the node IO throughput are the factors that have greater impact in the node local execution time, because ONE is IO-bound. To meet specific right time targets, the number of processing nodes has to be planned according to those targets.

Definition 5.4

Considering a ONE data model with the total data volume V, the local query execution time t_n in node i depends of the node's characteristics and the fraction of V that can be stored in node i, the $v_i = f(\mathrm{V})$. *For a homogenous deployment,* $v_i = f(\mathrm{V}) = \mathrm{V}/\eta$.

Since ONE does not require joins, the execution time *t(q)* of a query Q varies according to the data volume and the cost of filter and aggregation operations. Therefore, we can deliver accurate query execution time estimations. For any query *q,* we can provide right-time guarantees by adjusting the number of processing nodes and the data volume allocated to each node.

Since the performance of parallel shared-nothing deployments is constrained by the slowest node (Fig. 5.3-a), the amount of data allocated to each node (and how it is stored) should fully explore the nodes' characteristics, to evenly divide the processing time among nodes (Fig. 5.3-b). Data distribution, placement and join complexity (and processing cost) are all interrelated and dependent of the used data model. The simple addition of nodes to the parallel architecture is not sufficient to guarantee timely executions.

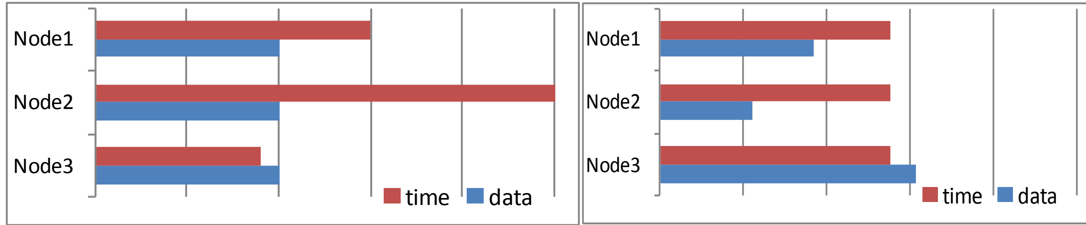


Fig. 5.3 – Data allocation size: a) evenly partitioned b) by nodes' performance

Our focus is in guaranteeing timely results for queries (including ad-hoc) by adapting the processing infrastructure, using an elastic parallel infrastructure where processing nodes are (de)allocated as required, the data placement and the data model, by redistributing the DW base data to provide query execution times within defined time targets (Fig. 5.4).

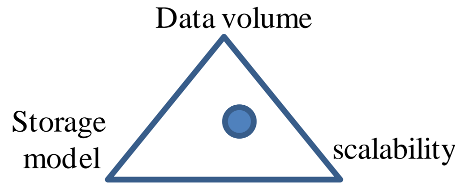


Fig. 5.4 – The timely execution triangle

5.3 TEEPA - Timely Execution with an Elastic Parallel Architecture

This section introduces TEEPA (a Timely Execution with an Elastic Parallel Architecture), a parallel architecture that provides timely execution results to queries submitted to DW schemas, by using an elastic set of heterogeneous nodes. It encompasses a set of modules that manage and seamlessly integrate an elastic set of heterogeneous nodes, and adjusts the data model and data volume allocated to each node according to the nodes' characteristics, in order to attain the required timely execution targets. TEEPA was designed as a transparent middleware and can be deployed with a wide range of heterogeneous nodes (illustrated in Fig. 5.5).

Definition 5.5

Consider an elastic parallel shared nothing infrastructure E composed with η *nodes,* $E=\{node_i\}_{i=1,\ldots,\eta}$ *and let* io_i *be the storage throughput of node i. Let P be a parallel shared nothing infrastructure composed with k nodes,* $P=\{node_k\}_{k=1,\ldots,k}$*, such that* $P\subseteq E$*. Consider* Q_r *the set of running queries,* $Q_r\subseteq Q$*, let* $t_{target}(Q_r)$ *be the tighter time target, such that target* $t_{target}(Q_r)=min\{t_{target}(q):\forall\ q\in Q_r\}$*. A parallel infrastructure is called elastic, iff for a different* Q_r' *and* $t_{target}(Q_r')$ *it*

can determine and compose a processing nodes P', P'⊆ E, and redistribute the data volume among P', such that t_{exec} *(*Q_r*', P')* $\leq t_{target}$ *(*Q_r*').*

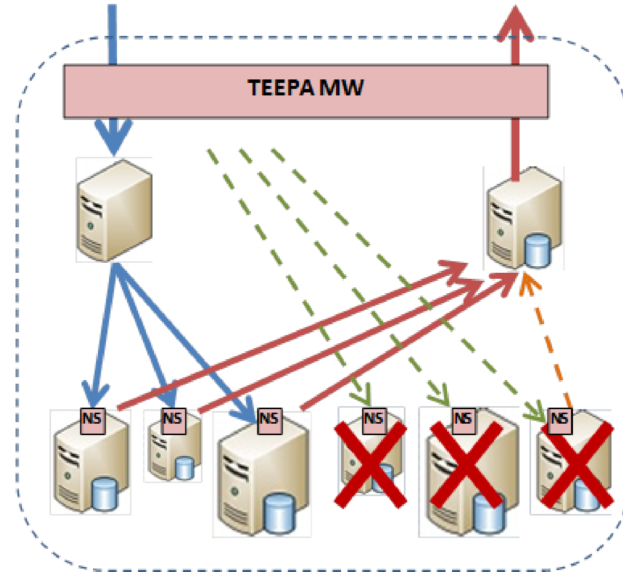


Fig. 5.5 – TEEPA over an elastic set of heterogeneous nodes

TEEPA follows a shared-nothing approach and has the ability to adjust the data model and query processing to fully explore the characteristics of added nodes that can be heterogeneous. The addition of more parallelism is performed with minimum disturbance and complexity, regardless of the new nodes characteristics. Willing nodes are registered with TEEPA middleware and their characteristics, such as available memory, processing and storage capacity, are accounted. When required, selected nodes are activated and after some data reorganization they are fully integrated within the processing infrastructure. This flexibility allows the use of an elastic set of heterogeneous nodes, that are added and/or removed as needed, in order to deliver timely execution targets. The additional nodes may include offline servers (to reduce energy costs) or other online non-dedicated nodes.

Definition 5.6

Let $t_{maxlocalexec}$ *be the maximum local execution time of each processing node,* $\forall i \in [1;\eta]$: $t_{maxlocalexec} \geq t_n\ (i,\ v_i)$. *The goal is to determine the number nodes* η *that are needed to guarantee that* $\forall i \in [1;\eta]$, $t_n\,(i,v_i) \leq t_{maxlocalexec}$, *and the data volume allocated to each node i,* $v_i = f(V)$.

TEEPA takes into account the user specified time targets, expressed at query or session level (see Section 5.3.1), to adjust and rebalance the processing infrastructure.

When the current deployment is unable to deliver the time targets, it adds more processing nodes and redistributes the data volume among them. These tasks are performed, respectively, by the Right Time Evaluator, the Node Allocation and Data Balancer sub-modules. TEEPA continuously monitors the local query execution, the IO throughput and the data volume allocated to each processing node, to determine if the system is able to satisfy the user specified time targets.

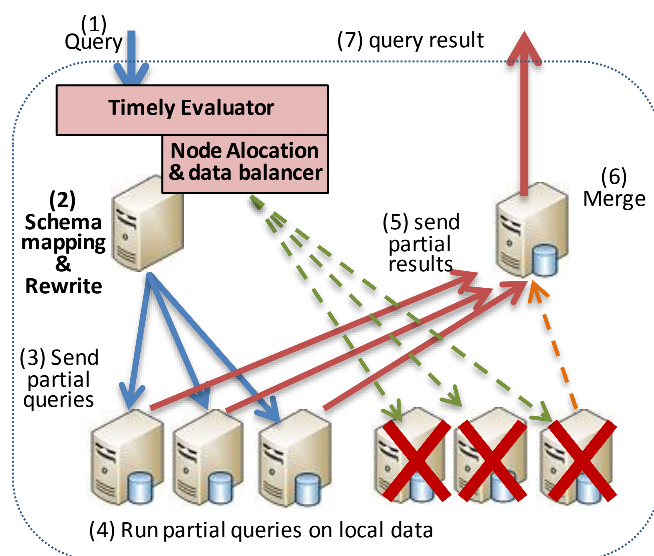


Fig. 5.6 – TEEPA framework

When TEEPA determines that a query may miss the time target, it starts the node allocation and data rebalancing process. TEEPA begins by determining the maximum data volume that a node can handle within the specified time target, and then it determines how many additional nodes are needed to process the whole data volume and satisfy the time targets. After the additional nodes are included in the parallel architecture, it emits a set of data reallocation tasks, indicating for each node the amount and the destination of the data to rebalance.

TEEPA is also designed to handle heterogeneous nodes. In this scenario, the data rebalancing tasks are issued according to each node capability. The data volume allocated to each node is adjusted as a function of the whole data load (total number of tuples), the tuple size and the scan throughput, with faster processing nodes handling larger data volumes. The node allocation (selection and integration of newer nodes) and data rebalancing tasks are continuously executed until the time targets can be assured (as illustrated in Fig. 5.7).

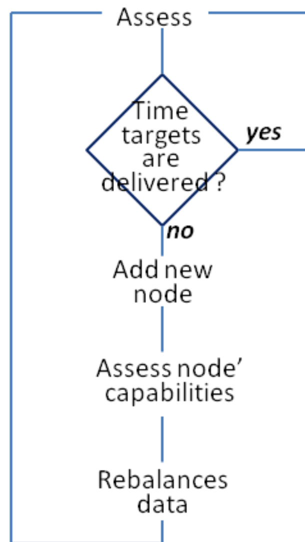


Fig. 5.7 – Data rebalancing process

Definition 5.7

Considering a ONE data model with the total data volume V, let io_i be the storage throughput (in MBps) of a node i. For a heterogeneous deployment, let tio be the aggregated storage throughput of active processing nodes, $tio = \sum io_i$, the data volume to allocate to a node i is, $v_i=f(V)=V\times (io_i / tio)$, i.e. the node i will hold a fraction of V that is proportional to its io throughput in the overall.

TEEPA integrates with a wide range of heterogeneous nodes and existing DBMS systems, and is composed by a set of interconnected and extensible modules. TEEPA is composed by two distinct services: the TEEPA manager, a managing service that handles the timely execution targets and manages the elastic set of processing nodes to ensure that the timely execution targets are delivered; and the TEEPA node service (TEEPA NS), a service that runs locally in each node and is responsible for locally processing the partial queries and also for processing all local management tasks related to the nodes' participation, or not, in the processing infrastructure. Fig. 5.8 depicts a detailed sketch of the TEEPA modules. The next sub-sections discuss in more detail their layers and functionalities.

TEEPA was conceived as a transparent middle-layer to users and applications, allowing a full compatibility with existing applications. TEEPA NS was designed to interact with the TEEPA manager, but it also can be deployed and usable as a standalone module, without the manager.

5.3.1 TEEPA Manager

The flexibility and scalability provided by the TEEPA architecture is achieved by the supervision of a TEEPA Manager, which processes all the necessary management tasks to adjust the processing infrastructure, by load-balancing data using an elastic set of heterogeneous nodes, to attain the timely execution targets. It handles the query timely execution requirements, manages nodes' availability and schedules the processing of tasks among nodes. It is divided in five replaceable components. Not all are mandatory in all deployments, e.g. a scalability manager is only required if the additional nodes can be added in runtime. Whilst the prototype version (illustrated in Fig. 5.8) uses a centralized management, TEEPA can also be run in decentralized manner for improved robustness and dependability purposes.

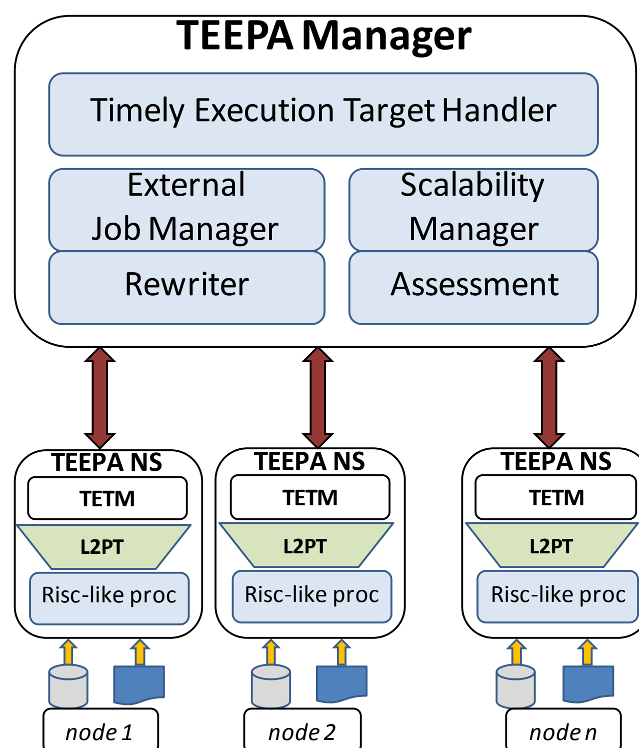


Fig. 5.8 – TEEPA modules in detail

TEEPA includes a module TETH which manages the time execution targets. The time execution targets are expressed through a set of additional clauses that are parsed and interpreted by TETH. Although TEEPA can be organized (dimensioned) to satisfy a global time execution target, it also allows the definition of a time execution target for a particular query (Fig. 5.9 shows the WITHIN clause of SELECT statement) or session (Fig. 5.10 shows the ALTER SESSION with time target clause). The timely

execution targets can be expressed as an absolute temporal instant, through the use of a AT clause (e.g. AT 3PM), or a relative time interval (e.g WITHIN 5 minutes).

```
SELECT   <attributes>
FROM     <tables>
WHERE    <conditions>
GROUP BY <grouping attributes>
[WITHIN  <value> {minutes | seconds} | AT <datetime> ]
```

Fig. 5.9 – Syntax of the time target clause added to a SELECT statement

```
ALTER SESSION SET TIME_TARGET
[WITHIN  <value> {minutes | seconds} | AT <datetime> ]
```

Fig. 5.10 – Syntax of the time target at session level

A submitted query is parsed to gather the time execution target. If it exists, then the query processing tasks are evaluated to assess if the time execution target can be satisfied. When it cannot, a data reorganization and rebalancing process is triggered (the scalability manager is signaled) in order to re-tune TEEPA parallel architecture according to the newly specified time execution target. Optionally, according to the management configuration, admission control actions can be employed (e.g. load shedding) when the current query load and processing infrastructure are unable to guarantee such time execution targets.

Each submitted query is rewritten by a **Query Rewriter**, into a set of parallel sub-queries (processing tasks) that are sent and run at each processing node. Additional sub-queries are also generated for merging (to aggregate) the partial intermediate results (computed locally in each node), to obtain the final result (the merging task). Since TEEPA acts as a transparent middleware, it always provides a consistent logical view of the star schema model to users and applications, without affecting the usability of existing applications. The rewriting process takes into account the node's data model, which may be different of the logical star schema model.

Sub-queries (processing tasks) and the merging tasks are scheduled by an **External Job Manager** which manages their orderly execution until the computation of the final results. Tasks are scheduled according to the nodes' availability, data placement, time execution targets and the currently running tasks. For dependability purposes, tasks may be redundantly scheduled for processing in different nodes containing data replicas.

TETH also collects the execution times of past queries, in order to determine if a new submitted query can execute within its time target. If is a known query, i.e. a query that have been executed in the past, it uses its query execution time do determine if this time is above the query target. For an unknown query, the maximum execution time of those past queries is used instead. TETH is modular and can be extended to include, for instance, the estimation of query execution costs provided by the underlying DBMS.

When TETH assesses that there is a high probability of missing the query execution targets, it signals a **Scalability Manager** to start the rebalancing process.

Deployment of a large star schema DW on complex and over-dimensioned clusters of parallel dedicated nodes may provide fast query results using an expensive brute-force approach. It follows the assumption that a large amount of top-edge hardware will be sufficient to have the job done. The over-dimensioning is lead by future performance and data load expectations, and by the fact that future upgrades are costly and may require a full architecture substitution, if similar (homogeneous) nodes became obsolete (unavailable). And a good balance of the amount of data allocated to heterogeneous nodes in a parallel shared-nothing architecture is hard to accomplish, since query time is constrained by the performance of the slowest node (see equation 2 in Section 5.1).

The **Scalability Manager** determines how many new nodes are required, to satisfy the demanded computational power, and the data volume that has to be moved from each node in order to achieve the specified query execution times (Section 5.6 discuss the overload detection and the rebalancing algorithms).

When additional nodes are available, or can be made available, they are activated, and after a data reallocation tasks have been processed, they are incorporated as processing nodes of the parallel infrastructure, as illustrated in Fig. 5.11.

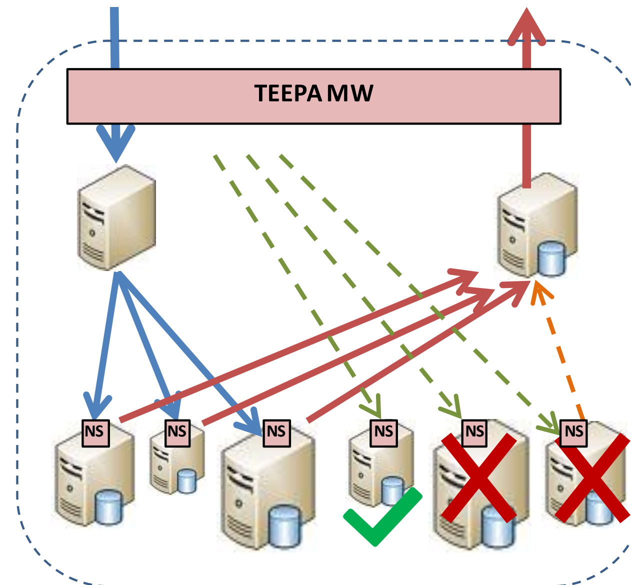


Fig. 5.11 – Additional node added

The amount of data has to be reallocated from current nodes to the newly added nodes and the detection of node unbalancing and overloading are discussed in detail in Sections 5.5 and 5.6 respectively.

After the rebalancing process is completed, it changes the status of these new nodes to online, before they became full members of the parallel infrastructure. This feature allows TEEPA to provide elastic processing capability and runtime load scalability. This module is also responsible for handling nodes availability, node failures, and for all the processing tasks related to the rebalancing of data and processing.

To provide top performance and perform informed rebalancing decisions, there is a continuous monitoring (**Assessment**) module that collects statistics, both at global and node level. Decisions taken by the TETH, the scalability handler and also the job scheduler are made taking into consideration the performance gathered statistics, and their quality.

5.3.2 Node Processing Service (TEEPA - NS)

Each processing node runs a small daemon, composed with three independent layers (shown in Fig. 5.8), that can be extended to include different implementations, as long as the interconnection interfaces are implemented. Not all layers may be required in all deployments. For instance, the TETM (defined below) can be excluded if its

functionalities are delivered by the TETH module. The same applies to the Logical to Physical Translator module (L2PT).

Query acceptance decisions can be made locally in each node, by a **Local Timely Execution Target Manager (TETM)**, through delegation or for queries that are submitted locally, outside the scope of the parallel execution infrastructure. Queries are analyzed and rewritten according to local data model, which may differ from the logical star schema data model, and are then scheduled according to the timely requirements. The local scheduling decisions also try to maximize the data sharing among running tasks.

TEEPA NS also includes an extensible Data Storage layer that abstract data processing tasks from the underlying storage organization, allowing the use of different storage systems, such as DBMS or raw file systems, in a seamless manner. This is possible since data processing is reduced to a minimal set of simple primitives.

To provide a seamless integration with users and applications, the architecture contains a **Logical to Physical Translator (L2PT)** module, which adapts the data physical organization, data placement and processing complexity to provide the required timely execution guarantees, while maintaining a consistent logical star schema view of the data. While the data model can change, by adapting the de-normalization level (fully or partially with small-sized dimensional data residing in memory), the logical view presented to users remains consistent and invariant. It provides a JDBC interface to allow seamless integration with users and applications and the submitted SQL queries are syntactically analyzed according to the logical star schema view.

5.4 Logical to Physical Translator

As the physical schema may diverge from the logical star schema view, queries have to be rewritten according to the physical schema representation, into a set of simpler processing tasks with more predictable execution time. A query q, syntactically valid according to the logical schema, is translated (rewritten) into query q' (or a set of sub-queries) according to the internal physical organization.

L2PT uses the partial de-normalization algorithms discussed in Section 4.8 to determine the level of data de-normalization and includes primitives that perform changes in the physical data representation, by varying the de-normalization level to guarantee timely execution guarantees. It also offers physical schema interface that transparently adapts to the query load. Multiple physical representations can coexist in each node while sufficient storage space is available. Each node can autonomously decide to change the de-normalization level to another model that best fits to the local data distribution and the timely execution requirements. The data model used in each node can fluctuate from the star schema data model to the fully de-normalized ONE data model, according to the node's characteristics (Fig. 5.12).

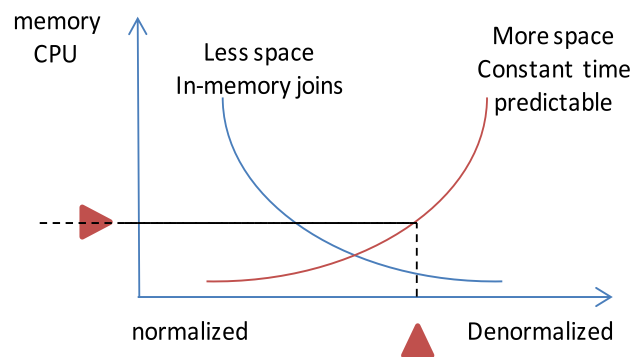


Fig. 5.12 – Storage space scope

An increase in the de-normalization level also increases the storage size requirements, since more redundant data has to be physically stored. However, all complex joins are removed (full de-normalization) or reduced to a minimal set of in-memory joins (partial de-normalization) where only dimensions that can fully reside in memory are left unchanged. Query processing requirements is thus reduced to a minimal subset of processing primitives without complex and costly join processing primitives. Queries are decomposed into a set of simpler and predictable processing instructions (see Section 4.1). Without costly joins (both in CPU and memory), and since it requires less memory for processing data, nodes change their memory management pattern by increasing the amount of memory used for caching the data, and consequently enhancing query performance even further.

5.5 Data Allocation to Heterogeneous Nodes

TEEPA manages an elastic set of heterogeneous nodes, added or removed according to the current timely execution targets, and re-balances data load between processing nodes until the variance of local query processing is below an acceptable threshold. Since ONE-P does not require joins, it provides predictable and almost invariant query execution times, with less demanding DBMS systems, and therefore it can be deployed on COTS (see acronym) hardware with minimal memory and processing requirements. [Costa, Cecílio, et al. 2011; Raman et al. 2008] demonstrated that schema de-normalization provides predictable query execution times. Since ONE-P provides a high degree of predictability, and query processing is highly IO dependent, the de-normalized relation can be freely partitioned according to each node's capacity.

Query processing of a de-normalized relation in a shared-nothing fashion, illustrated in Fig. 5.13 b) for active nodes is similar to typical shared nothing infrastructures. The main difference in query processing resides on step 2, which rewrites a query into a set of partial queries to be executed by the local nodes. Besides rewriting, it also has to remove all join conditions and perform the necessary attribute mapping, translating all the existing references of the star schema model to the corresponding attributes in the de-normalized relation. To ensure that dimension aggregates results do not include double-counting, some additional predicates have to be included**.**

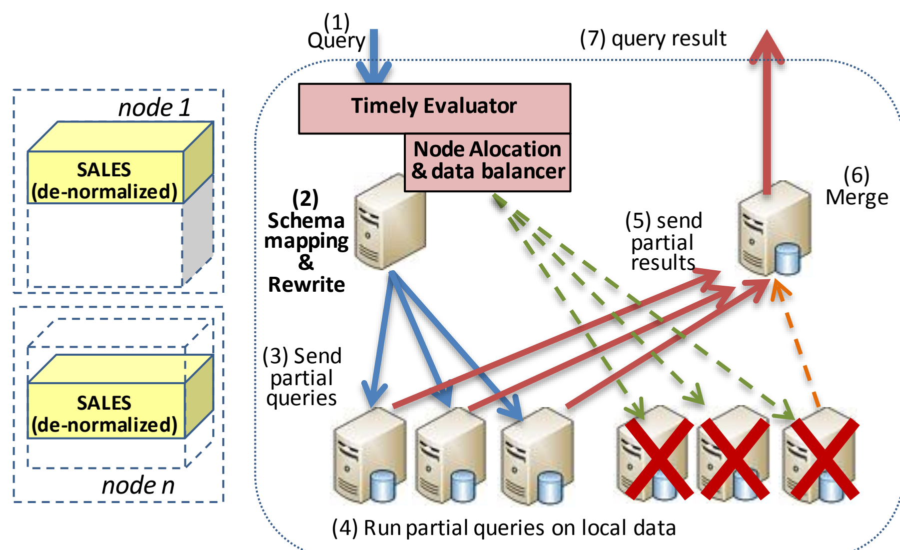


Fig. 5.13 – a) ONE-P partitioning and placement b) ONE-P Query processing

In a parallel deployment with homogenous nodes, the amount of data is evenly divided between nodes to deliver the $t(Q)$ expected execution time. The number of nodes η is determined as a function of the desired timely execution, so that

$$t_{\mathrm{n}}(Q_p) = t(Q) - t_{tpr}(Q_{pR}) - t_{m_j}\left(\sum_{j=1}^{\eta} Q_{pR}\right) \quad (3)$$

However, in parallel architecture with heterogeneous nodes, and since the ONE is highly IO dependent, data is allocated to each node in order to minimize the variance of the inter-node local query execution time.

$$max\left(\left\{t_{n_j}(Q_p)\right\}_{j=1}^{\eta}\right) = t(Q) - t_{tpr}(Q_{pR}) - t_{m_j}\left(\sum_{j=1}^{\eta} Q_{pR}\right) \quad (4)$$

The maximum local query execution local time (equation 4), is adjusted when the number of processing nodes increases, to account for the additional intermediate results that have to be exchanged and merged before computing the final result.

5.6 Detection of node unbalancing and overloading

Local execution time is tightly related, not only to the node's physical characteristics, but also to the data volume that the node has to process. Therefore to get a local processing execution time below a given maximum local execution time *(max_ltime),* it is fundamental to reduce the data volume allocated to each node according to its capability.

Algorithm: *exceeding_load*
For a given maximum local execution time, determine the exceeding data load that has to be removed and distributed among nodes

Input: *on_nodes* – list of online nodes
Input: *max_ltime* – maximum execution local execution time
Output: *exceed_nodes* – list of overloaded online nodes with exceeding data
Output: *under_nodes* – list of online nodes with underused data
Output: *exceedload* – amount of exceeding data volume in online nodes
1 **if** *on_nodes* = ∅ **then** return 0;
2 **Let** *exceedload* ←0;
3 **foreach** *node* **in** *on_nodes* **do**
4 | **Let** *ntuples* ← *getNumberofTuples* (*node*);
5 | **Let** *nltime* ← *exec_time_history* (*node*);
6 | **Let** *nltps* ← *nltime* / *ntuples*;
7 | **if** *nltime* > *max_ltime* **then**

```
8          Let exceed_nodes (node) ← (nltime - max_ltime) × nltps;
9          Let exceedload← exceedload + (nltime - max_ltime) × nltps;
10     else
11         Let under_nodes (node) ← (max_ltime - nltime) × nltps;
12 return exceedload;
```

When the *exceed_nodes* list is not empty, it means that some of the processing nodes are overloaded and are unable to timely process their local data, and therefore some sort of data rebalancing is required. The overloading can be resolved by redistributing the exceeding data of the overloaded nodes among the other nodes (if exists) or by the allocation of additional nodes.

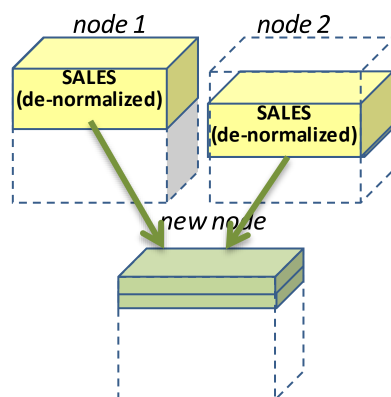


Fig. 5.14 – Rebalancing the data volume when a new node is added

Fig. 5.14 illustrates the data rebalancing process, with each node determining and distributing the exceeding data volume into the new node that was added to the parallel infrastructure.

5.6.1 Resolving overloads by rebalancing data among nodes

The Timely Evaluator is continuously monitoring and assessing if the current ONE-P deployment can deliver the user-specified time targets. When it cannot and there are overloaded nodes, the Data Balancer determines if the non-overloaded nodes can handle the exceeding data without endangering their own local execution targets. The *redistribute_exceeding_load* algorithm depicted below, starts by rebalancing data chunks from most overload nodes into under loaded nodes. When that occurs, the data chunk is de-registered from the registry of the overloaded node, and a new registry entry is inserted in the destination node.

Algorithm: *redistribute_exceeding_load*
Redistributes the exceeding data load between the non-overloaded nodes without overloading them.

Input/Output: *exceed_nodes* – list of online nodes with exceeding data (size)
Input/Output: *under_nodes*– list of online underused nodes with capacity (size)

```
1  while exceed_nodes != ∅ && under_nodes != ∅ do
2    sort_descending_by_exceeding_load ( exceed_nodes ) ;
3    sort_descending_by_under_load ( under_nodes ) ;
4    Let enode = exceed_nodes [0];
5    Let unode = under_nodes [0];
6    Let ntuples ← MIN ( load( unode ) , load( enode ) );
7    rebalance_tuples_between ( enode, unode, ntuples) ;
8    if  load( enode ) > load( unode ) then
9      Let exceed_nodes [0] ← exceed_nodes [0] - ntuples;
10     Let under_nodes ← under_nodes – { unode };
11   else if  load( enode ) < load( unode ) then
12     Let under_nodes [0] ← under_nodes [0] - ntuples;
13     Let exceed_nodes ← exceed_nodes – { enode };
14   else
15     Let under_nodes  ← under_nodes   – { unode };
16     Let exceed_nodes ← exceed_nodes – { enode };
```

To fine-tune the amount of data allocated to each node, according to its characteristics, the data reallocation process can be performed at row-level.

5.6.2 Resolving overload by allocation of additional nodes

TEEPA maintains a registry of the heterogeneous processing nodes, which can be activated when additional processing power is required to deliver stricter timely query results. The Timely Evaluator module is continuously monitoring and assessing if the current ONE-P deployment can deliver the user-specified time targets. When it cannot be guaranteed, the Node Allocation & Data Balancer module determines the maximum data volume that can be timely processed by each node (within the time targets), and redistributes the remaining data to the new nodes. By increasing the number of processing nodes, the data volume allocated to each node is reduced, and consequently the local execution time. However, as we increase the number of processing nodes, there is also an increase in time needed for exchanging and merging the intermediate results. Therefore, the rebalancing process reduces the maximum local execution target in each node to accommodate these increased costs. To minimize the exchanging time and the number of nodes present in the parallel processing

infrastructure, the new nodes are selected according to their performance (algorithm *allocate_fastest_first* shown below). The node's performance is mainly based on IO aspects.

Algorithm: *allocate_fastest_first*
Allocate new nodes, fastest nodes first

Input: *on_nodes* – list of online nodes
Input: *off_nodes* – list of offline nodes
Input/Output: *exceed_nodes* – list of online nodes with exceeding data (size)
Input/Output: *under_nodes* – list of online nodes with under data (size)
Input: *max_ltime* – maximum execution local execution time

```
1 sort_descending_by_IO ( off_nodes ) ;
2 foreach nnode in off_nodes do
3 |  activate ( nnode );
4 |  Let on_nodes ← on_nodes ∪ { nnode };
5 |  Let max_ltime ← max_ltime – merge_cost ( nnode );
6 |  Let exceed_load ← exceeding_load ( on_nodes , max_ltime) ;
7 |  if exceed_nodes =0 then break;
8 redistribute_exceeding_load ( exceed_nodes, under_nodes );
```

The *allocate_fastest_first* algorithm may not deliver the fastest reorganization time, since its goal is to minimize the number of processing nodes. If the number of processing nodes is not an issue, but the reorganization time is, we use a variant of the *redistribute_exceeding_load* where the function that sorts non-overload nodes (*sort_descending_by_under_load* (*under_nodes*)) is replaced by similar function the *sort_ascending_by_under_load_transfertime* (*under_nodes*) that takes into account the nodes' IO performance and also the network interface performance. In this case, nodes that exhibit less transfer times are used first.

While this process is performed, the Timely Evaluator continuous to monitor the current deployment to assess if the time target is provided, repeating the process until the average inter-node variance falls below a given threshold. The node allocation (selection and integration of new nodes) and the data rebalancing tasks are continuously executed until the time targets are assured.

After the rebalancing process is completed, the Node Allocation process changes the status of these new nodes to online before they start to be full members of the parallel infrastructure. It is also responsible for handling nodes availability, and for all the rebalancing data and processing tasks that are required to handle a node failure.

To provide top performance and perform informed rebalancing decisions, there is a continuous monitoring and collection of statistics, both at global and node level.

5.7 Chapter Summary

We proposed TEEPA, a parallel right-time architecture that uses the particular characteristics of the ONE model. TEEPA provides predictable performance to scalable data volumes and can be massively parallelized over a large set of processing nodes.

We describe the modules of the architecture, and the mechanisms that determine when the current deployment is unable to provide timely execution times, and determines how many, and which nodes, have to be integrated into the parallel deployment. It also determines the optimal data layout and the data volume that has to be redistributed among nodes to provide specific execution time targets.

Chapter 6

SPIN: Concurrent Workload Scaling over Data Warehouses

Increasingly, Data Warehouse (DW) analyses are being used not only for strategic business decisions but also as valuable tools in operational daily decisions. As a consequence, a large number of queries are concurrently submitted, stressing the database engine ability to handle such query workloads without severely degrading query response times. The query at a time execution model of traditional RDBMS systems, where each query is evaluated as a separate execution plan, does not provide a scalable environment to handle such increase of unpredictable workload. While the previous chapter focuses on providing guarantees for varying data volumes, this chapter proposes strategies to provide predictable and time guaranteed execution for highly concurrent query workloads.

This chapter proposes SPIN, a data and processing sharing model that delivers predictable execution times to aggregated star-queries even in the presence of large concurrent query loads, without the memory and scalability limitations of existing approaches.

Section 6.1 describes the SPIN processing model and Section 6.2 describes how it overcomes concurrency limitations of the query-at-time processing model of common database engines. Section 6.3 describes the mechanisms used by SPIN to embed data and queries into a shared query processing pipeline tree and Section 6.4. discuss the query processing path and how SPIN dynamically reorganizes the processing tree. We discuss how SPIN characteristics overcome the limitations of recent proposals on data and processing sharing, such as memory limitations, reduced scalability and

unpredictable execution times when applied to large DWs, particularly those with large dimensions.

Section 6.5 describes how data is processed and flow in the workload processing tree (*WPtree*) in order to maximize the sharing of data and processing, and Section 6.6 describes how this tree is built and reorganized when new queries are submitted.

Section 6.7 discusses how SPIN handles data updates and deletes. Section 6.8 presents implementation details of the SPIN prototype. The chapter ends with some concluding remarks.

6.1 SPIN Processing Model

SPIN uses the de-normalized data model (ONE) proposed in [Costa, Cecílio, et al. 2011; Costa, Martins, et al. 2011], this means that the star schema is physically organized as a single de-normalized relation (O_d). SPIN provides workload scale-out by combining (merging) data requests from all queries to be satisfied by a sequential continuous scan executed in a circular loop. Data is read from storage and shared to all running concurrent queries, as illustrated in Fig. 6.1.

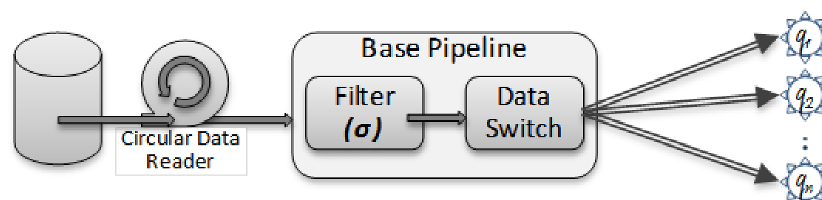


Fig. 6.1 – SPIN base data processing model

The circular loop is continuously spinning, sequentially reading data chunks, while there are queries running. The execution of a query ends when all tuples stored within data fragments are processed and the circular logical loop is completed.

A Data Reader sequentially reads chunks of relation O_d, and continuously fills a data pipeline. It views the ONE relation (O_d) logically as a circular relation, i.e. a relation that is constantly scanned in a circular fashion (when the end is reached, it continues scanning from the beginning). The relation is divided into a set of logical fragments (or chunks), with the chunk size ranging from 32MB to 512MB in size, in multiples of 64 tuples, adjusted to storage characteristics. A fragment metadata repository is maintained storing information regarding each data fragment, namely its

size, its start and end logical positions within the relation and the number of tuples that it stores. Fig. 6.2 illustrates the SPIN metadata repository which stores metadata information about the relation and data fragments.

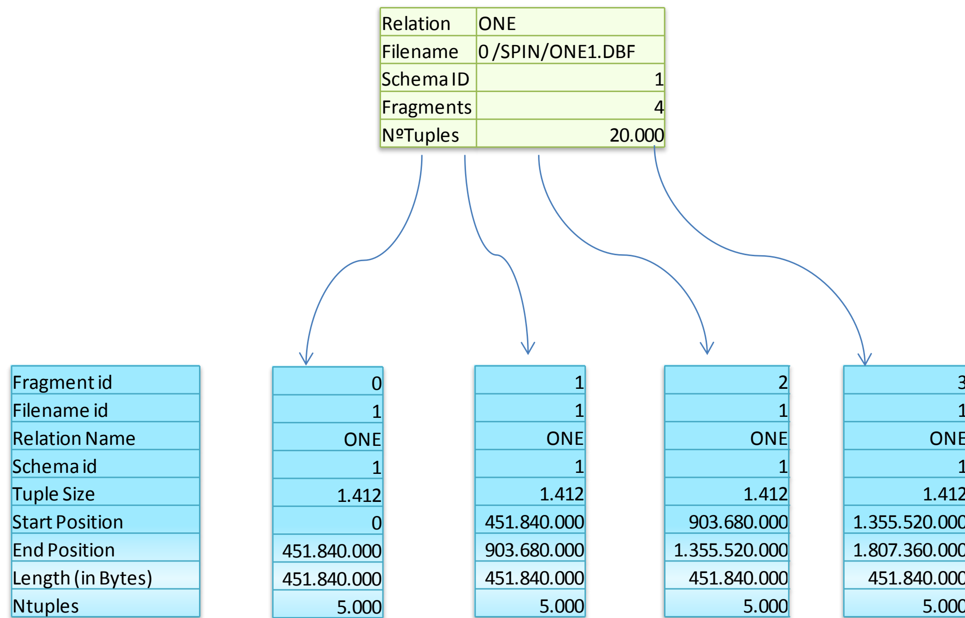


Fig. 6.2 – SPIN Fragment Metadata

A Data Reader sequentially reads chunks of relation O_d, and continuously fills a data pipeline. When entering the pipeline, tuples have to go through a fast initial filtering selection operator to early discard large subsets of tuples not requested by currently registered queries (early selection). Only tuples that are relevant for at least one query flow to a Data Switch (*DS*), which diverts tuples to each running query, building a dedicated logical flow (data branch) for each query. For performance reasons, some fast execution and common early selection predicates are incorporated in the Data Reader.

Definition 6.1

Let $|O_d|$ *be the number of tuples of relation* O_d. *For a Data Reader DR that is continuously reading* O_d *in a circular fashion,* $r(O_d)$ *is the index of the last tuple read and placed into the pipeline, with* $r(O_d)$ *ranging in* $[0; |O_d|]$.

Since O_d is de-normalized, no costly join tasks need to be processed, only query predicates and aggregations operations. Any query q, when submitted, starts consuming and processing the tuples that are currently in the data pipeline, starting from $r(O_d)$.

6.2 SPIN query handling

A Query Handler handles query (de)registering. Any query q registers as a consumer of the data in the pipeline, following a publish-subscribe model. Each query q, has to process every tuple from relation O_d, at most once, although data query processing does not need to start at record 0.

Definition 6.2

Let Q_r be the current query workload, composed by a set of queries, $Q_r = \{q_i\}$, $q_i \in Q$, and let $|Q_r|$ be the number of running queries. A query q_j, $q_j \in Q$, when submitted it is added to Q_r and the current value of $r(O_d)$ is registered. Let $s(q_j)$ be the index of the first tuple to be considered for processing query q_j, then $s(q_j) = r(O_d)$.

For each running query, a Query Handler maintains an indicator of the first logical tuple (position in the circular loop) consumed by the query. This logical first row position is fundamental to determine when the end of each query is reached. When that occurs, then the query q has considered all tuples for execution, therefore it terminates execution and sends the query results to the client. Fig. 6.3 illustrates the data reading circular process, depicting the logical starting position for queries q_1, q_2 and q_3.

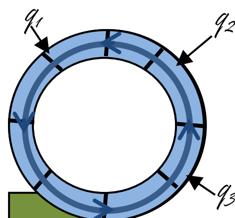


Fig. 6.3 – SPIN sequential data reading loop

Since each query q, starts consuming tuples from the current position ($r(O_d)$) in the circular loop, without the need to start from a specific record, the reading cost (IO cost) is shared among all running queries Q_r without introducing additional IO overhead or random reads. Others costs related to query processing occur at subsequent executing phases, such as selection, logical branching and pipeline processing.

6.3 SPIN operators and data processing pipelines

SPIN follows a flow oriented processing model. Each query *q* is analyzed and decomposed into a set of sequentially-organized predicates, computations and aggregations tasks, which are mapped to operators placed along pipelines. A data processing pipeline (or simply referenced as pipeline) is a collection of sequentially-organized operators that transform the input tuples as they flow along the pipeline.

SPIN include the following base operators:

- **Selection Operators -σ (***conditions***)** – apply predicate clauses to filter incoming tuples, trashing those that do not meet the predicate clauses. Each Selection Operation maps a query predicate (default) or a set of related query predicates. The selection operators are typically placed at the beginning of the processing pipeline, to filter tuples that pass-through the pipeline and go into subsequent processing operators. Selection Operators are placed in a sequential ordered fashion according to their selectivity and evaluation costs, with more restrictive and faster ones placed at early stages.
- **Projection operators -π (***attributes***)** – restrict to a subset of tuple attributes that flows throw the pipeline. A projection and a selection operator can be combined into a single step operator.
- **Computation Operators -φ (***computation expression***)** - perform tuple level data transformations, including arithmetic (e.g. φ (a + b)) and string manipulation (e.g. substring). A dedicated Computational Operator is built for each specific transformation. A Computation Operator maps a tuple-level arithmetic function or operation expressed in any of the query clauses (e.g. the arithmetic expression `QUANTITY×PRICE`). The mapping into φ is particularly relevant for complex computations that appear in several query clauses. Query predicates that include computations (e.g. `QUANTITY×PRICE>1000`) may be mapped into σ, preceded by a φ that performs the computation. The goal is to build fast selection operation with simple and fast evaluation predicates.
- **Data Switches - $\mathcal{DS}$ (***attribute conditional switching***)** – forward data tuples into a set of data outputs, called **logical data branches ($\mathcal{B}$)**. Tuples are forwarded to all branches *b, b* ϵ *B*, or forwarded according to each branch conditions. For

each branch, a tuple is only forwarded if it matches the branch's conditions b_p (if exists). Tuples not matching any branch predicates are trashed.

- **Aggregation operators – Σ** (*grouping attributes; aggregation functions*) – perform group by computations, by grouping tuples according to GROUP By clauses, and applying aggregation functions (e.g. SUM). Aggregation operators output results when all tuples for a given query as been considered for processing.

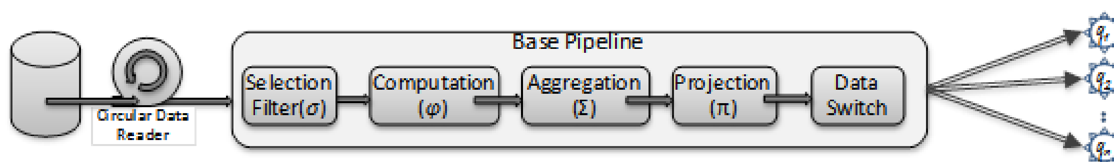


Fig. 6.4 – SPIN Data pipeline with operators

The initial pipeline, where the data is placed by the Data Reader, illustrated in Fig. 6.4, is named the base (or root) pipeline p_b.

When submitted, a query q is analyzed and decomposed into a sequentially-organized set of predicates, computations and aggregations tasks, which are then mapped into SPIN operators and placed along a query-specific pipeline, or $p_s(q)$. The processing of query q only starts after $p_s(q)$ is built and registered as a data consumer of the base pipeline p_b.

Definition 6.3

For a query q, $q \in Q$, $p_s(q)$ is the query specific pipeline containing the query specific operators.

Since each $p_s(q)$ consumes the data from the base pipeline p_b, the cost of gathering it from storage is constant and shared among queries, regardless of number of running queries $|Q_r|$. However, as $|Q_r|$ increases there is also an increase in number of operators and query-specific pipelines (one for each query).

6.4 Query processing path

In large concurrent query loads, subsets of queries, $Q_s \subseteq Q_r$, may have the same query predicates, computations or aggregations operators, which are processed concurrently exhausting memory and processing resources. SPIN, to avoid the redundant processing and to enhance the sharing of data and processing, splits the query-specific pipeline $p_s(q)$ into an equivalent ordered set of sequentially connected pipelines $\{p_1,\dots,p_n\}$. Each pipeline is composed with one or a set of logically related operators that have to evaluate the same incoming tuples, and is connected as a data consumer of its predecessor. Operators that evaluate distinct sets of tuples are placed in distinct pipelines, e.g. the selection operators $\sigma_{(year=2000)}$ *and* $\sigma_{(product='P1')}$ are placed into distinct pipelines since they have different selectivities. This set of sequentially connected pipelines is called the query execution path, or *path*(q), and its output result is equivalent to $p_s(q)$, $path(q) \Leftrightarrow p_s(q)$.

Definition 6.4

A data processing pipeline p is a set of sequentially organized operators that transform tuples as they flow along the pipeline. Let p_i and p_j be two distinct pipelines, $p_i \rightarrow p_j$ denote that p_j is directly connected to p_i and consumes the results of p_i. For a query q, $q \in Q$, let $path(q)=\{p_1,\dots,p_n\}_{,n\geq 1,}$ denotes a set of sequentially connected pipelines that form the query path, such that

$\forall\, p_{i,i>1 \wedge i \leq n},\ p_i \rightarrow p_{i-1}.$

For the currently running query load, similar partial pipelines (with the same operators over the same tuples) from different queries are combined into a common pipeline and a data switch is appended at the end to share its results. The subsequent connected data pipelines are then connected as logical branches of this common data pipeline, consuming its output.

Definition 6.5

A logical branch b is a pipeline, $b\in path(q)$, that is connected to a common pipeline p_c, or to the base pipeline p_b. For a set of queries $Q_s=\{q_i\}$, $Q_s \subseteq Q_r$, with $|Q_s|>1$, when exists a pipeline p_c, such

$\forall q\in Q_s$, $\exists p'\in path(q)$: $p_c \Leftrightarrow p'$ *and* $\exists b\in path(q)$: $b\neq p' \wedge b\rightarrow p'$,

then p_c is a common pipeline, b is a branch and $path(q)= \{b\}\cap path'(q)$: $\{b\}\rightarrow path'(q)$, $path'(q)= path(q) \setminus \{p'\}$.

A set of query-specific pipelines with common operators are rearranged in order to push–forward and orchestrate similar operators into a common processing pipeline. At the end of this pipeline, a data switch *DS* diverts the data output to further processing in subsequent logical branches. Fig. 6.5 illustrates a SPIN processing layout with two logical branches composed by two query-specific processing pipelines connected to a common processing pipeline.

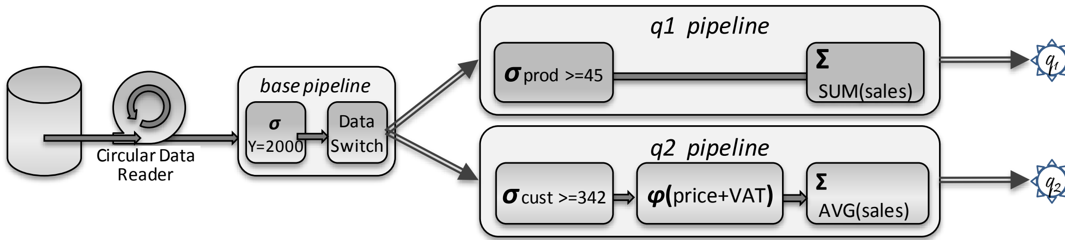


Fig. 6.5 – SPIN Logical Branch Processing Model

Definition 6.6

For a query q, $q\in Q_s$, let b_n be the logical branch with the query-specific operators and p_b the base pipeline, then for $\forall q\in Q_r$, the path of a query q can be expressed in the form of $path(q) = \{ p_b, b_1, ..., b_n \}_{,n\geq 0}$.

For the current query load Q_r, when exists a pipeline p that is common to all running queries, then the operators are pushed forward and combined with the base pipeline p_b, in order to early discard unnecessary data and reduce the data volume that is placed in the base pipeline p_b. $\forall q \in Q_r$, $\exists b\in path(q)$, then $p_b = p_b \cup b$. When possible, selection predicates are pushed forward into the Data Reader to reduce the data volume within the pipelines and thus increasing the level of data processing sharing. The base pipeline trashes tuples that are not required by any of its logical branches. To ensure a fast early selection phase, complex (costly) query predicates are placed at later stages.

6.5 Building the Workload Processing Tree

As a result, the initial query-specific processing pipelines of running queries are split, merged and organized into a workload data processing tree. In the end, for each query there will be a logical data path traversing logical branches and pipelines that is equivalent to the initial query-specific data pipeline.

Definition 6.7

For a query workload Q_r, the orchestration of all the query paths, such that $\forall q \in Q_r$ with $path(q) = \{ p_b, b_1, ..., b_i, p_q \}$, in a tree manner with p_b as the tree' root, is named the workload processing tree, or WPtree. For a pipeline p of the workload processing tree, $\forall p \in$ WPtree, let $\beta(p)$ denote the set of branches that are connected as consumers of p.

The merging of common data processing pipelines is enforced through all the query execution steps, to maximize the sharing of data and processing and consequently reduce memory and processing requirements. For instance, consider that queries q_1, q_2 and q_3 are currently running with different query predicates: q_1 ($\sigma_{p=a}$), q_2 ($\sigma_{y=2000}$) and $q_3(\sigma_{p=a \wedge y=2000})$. Without combining the common pipelines of the running queries, the initial query-specific pipeline p_s is built and connected to the base pipeline. Fig. 6.6 depicts the query-specific pipeline, one for each query. In the example we observe that some selection and aggregator operators can be combined into common pipelines.

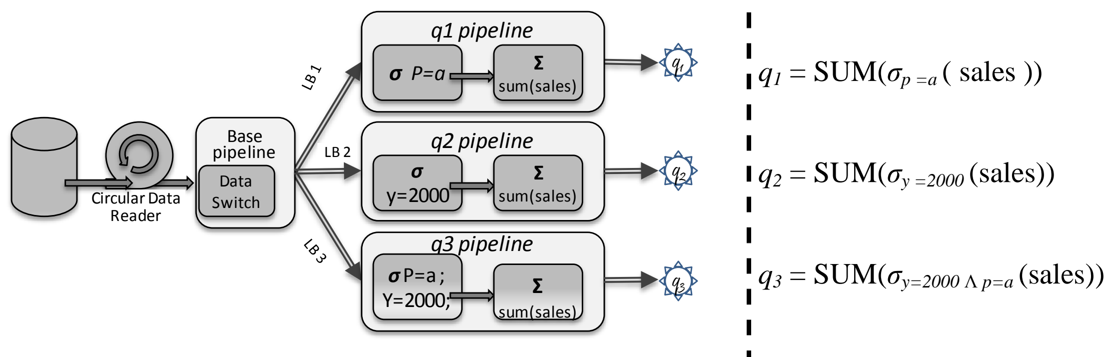


Fig. 6.6 – SPIN deployment of query-specific pipelines

When a new query is submitted, and instead of simply connecting the corresponding query-specific pipeline to the base pipeline, we apply a merging algorithm that tries to find interception points in the current workload processing tree, that can be combined in order to maximize the sharing of data and processing of running queries. The ***addPath*** algorithm, shown below, uses the query path of the

newly submitted query to update the workload processing tree to reflect the changes introduced by the merging of common processing pipelines.

Algorithm: *addPath*
Adding a query-specific pipeline to the Workload Processing Tree

---

Input: Q_{path} the query-specific pipeline of query q: $Q_{path} = path(q)$
Input: p a WPtree's pipeline, by default the base (root) pipeline, p_b
Data: Workload Processing Tree (*WPtree*)

```
1  if Q_path = ∅ then return addBranch ( p, Q_path);
2  Let β be the branches connect to p; i.e. next connected branches
3  if ∃ b ∈ β , ∃ p' ∈ Q_path : p' ⇔ b then
4  |   Q_path ← Q_path - {p'};
5  |   return addPath ( b, Q_path );
6  if ∃ b∈β ∧ p'∈Q_path : p' ∩ b ≠ ∅ then
7  |   p_common ← b ∩ p';
8  |   p_rest ← b \ p_common;
9  |   Q_path ← Q_path \ {p'};
10 |   addPath ( p, Q_path + {p_rest});
11 |   return addPath (b, Q_path );
12 return addBranch ( p, Q_path);
```

The algorithm uses the query path, $Q_{path}=path(q)$, the set of sequentially connected pipelines, and for each pipeline p starting from the base (or root) pipeline p_b, of the current workload processing tree (*WPtree*) determine if exists a logical branch b that matches a pipeline p' of Q_{path}. If exists then p' is removed from Q_{path} and the remaining Q_{path} is connected as a logical branch of b and consumes its data output. This process is applied to each of *WPtree*'s pipelines, starting from the WPtree's root pipeline to the leaves, while there is a *WPtree*'s pipeline (b) that fully matches a pipeline of Q_{path}. The matching pipelines are removed from Q_{path}.

Otherwise, if Q_{path} cannot does not have a pipeline p' that completely matches a branch b, but there is a partial match between a *WPtree*'s pipeline b and a Q_{path}'s pipeline p, meaning that they share a common region (interception between their selection predicate regions), then Q_{path} is divided into $Q_{path1} = Q_{path} - \{p\}$ and $Q_{path2} = Q_{path1} + \{p \cap b\}$. The algorithm is then applied separately to each of those paths.

When do not exists a branch b of pipeline p that matches any of Q_{path}'s pipelines, either fully or partially, then Q_{path} is connected as a new branch of the last matching *WPtree'* pipeline p. If no matching pipeline exists then Q_{path} is connected to the base pipeline.

The new logical branches are connected through a *DS* in the last matching pipeline. The number and placement of *DS*s, and logical branches, are orchestrated in order to minimize the switching cost (DS_{cost}), the number of evaluated predicates (n_{eval}), the predicate evaluation costs and the memory requirement for branch management.

The placement of *DS,* and **σ,** along the *WPtree* influences the data volume that goes through the pipelines. Therefore, we apply an optimization process that traverses all logical branches to find opportunities to push-forward operators into preceding pipelines. For instance, if every branch of a pipeline has the same operator, this operator can be moved into the pipeline. Within a pipeline, selection operators are placed at the beginning. Whenever possible, particularly for fast selection operators, a selection operator within a branch is replaced by a data switch condition at the DS operator of the preceding pipeline. For instance, in Fig. 6.7 the selection operators of pipeline 3 and 4 where pushed forward to pipeline 1 as a DS. The goal is to reduce the data volume that traverses the pipelines, and to maximize data and processing sharing.

SPIN applies a merging process that analyses selection predicates and how processing and intermediate results can be shared among processing pipelines. For each data path $path(q)$, it follows the path backward to p_b, and at each pipeline $p \in path(q)$, it determines if there exists other logical path that is processing, or has already processed, a subset of the tuples that p has to process. When a logical path *lpath* exists, then p is split into two sequential pipelines (p_1 and p_2) with p_2 containing the *DS* of p, preceded by a merging operator. p_2 is connected to p_1 and *lpath* and starts consuming their outputs and merging the results. The selection predicates of p_1 are updated to exclude the predicates of the logical path *lpath*. This can result in multiple alternative branching deployments.

To evaluate these alternative deployments, and merging configurations, the merging process uses several data volume metrics:

- n_{tuples} as the number of relation tuples,
- n_{eval} as the total number of evaluated tuples by the **σ**s, and
- n_{ag} as the total number of aggregated tuples by the **Σ**s.

In the example, the initial deployment, without considering merging, the number of evaluated and aggregated tuples are computed, respectively, as

$$n_{eval} = 3 \times n_{tuples}$$

$$n_{ag} = n_{eval}\,(\sigma_{p=a}) + n_{eval}\,(\sigma_{y=2000}) + n_{eval}\,(\sigma_{p=a\,,\,y=2000})$$

After the merging process, the number of evaluated tuples n_{eval} which has obtained as a function of the number of running queries Q, is reduced from $Q \times n_{tuples}$ to $\Sigma n_{eval}(p_\sigma)$ with p_σ the selection predicates of p. Fig. 6.7 depicts the final deployment after the merging process.

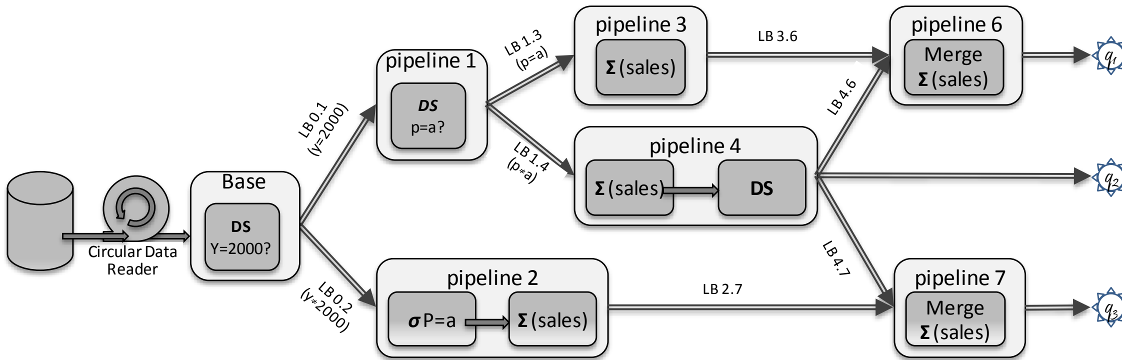


Fig. 6.7 – Aggregation Branch processing

The number of aggregated tuples n_{ag} is also reduced from $\Sigma n_{eval}\,(\sigma_q)$ to $n_{eval}\,(\sigma_{y=2000}) + n_{eval}\,(\sigma_{p=a\,,\,y=2000})$. The total number of evaluated and aggregated tuples are computed, respectively as

$$n_{eval} = n_{eval}\,(\sigma_{y=2000}) + n_{eval}\,(\sigma_{y=2000}) + n_{eval}\,(\sigma_{y\neq 2000}) = n_{tuples} + n_{eval}\,(\sigma_{y=2000})$$

$$n_{ag} = n_{eval}\,(\sigma_{p=a\,,\,y=2000}) + n_{eval}\,(\sigma_{p=a\,,\,y\neq 2000}) + n_{eval}\,(\sigma_{p\neq a\,,\,y=2000})$$
$$= n_{eval}\,(\sigma_{y=2000}) + n_{eval}\,(\sigma_{p=a\,,\,y\neq 2000})$$

In this example, we observe that the number of evaluated tuples (n_{eval}) is substantially reduced from 3 times the number of tuples (n_{tuples}) to n_{tuples} plus the number of tuples that satisfy the predicate ($\sigma_{y=2000}$). More than 1/3 of the tuples, depending of the selectivity of $\sigma_{y=2000}$, aren't evaluated. This reduction is even greater as the number of concurrent queries increases and as the overlapping of query predicates increases. Fig. 6.8, extracted from Chapter 8, illustrates this behavior.

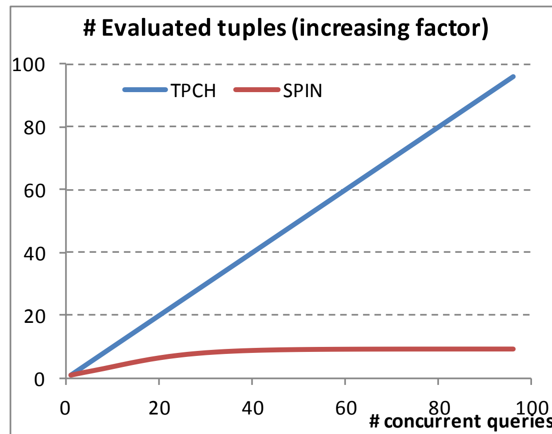


Fig. 6.8 – Number of times tuples are evaluated

The results show, as the number of concurrent queries increases, a significant reduction in the number of evaluated tuples by SPIN, while observing an almost linear increase in the number of evaluated tuples of the common query-at-time processing model of most database systems.

6.6 Reorganization of the workload processing tree

The data branching deployment is continuously reorganized as new queries are submitted. New queries that cannot be directly plugged into the current workload processing tree *WPtree*, as discussed in Section 6.5, are maintained as distinct branches of the base pipeline p_b.

When a query finishes its execution, the Query Handler removes from the workload processing tree the query-specific pipelines that are not being used in other logical paths. Pipelines used by other logical paths are maintained and only the query-specific logical branches are detached and removed. For instance, in the example illustrated in Fig. 6.7, when the query q_3 finishes its execution, the corresponding query-specific pipeline (pipeline 7) is detached from pipelines 2 and 4 before being removed.

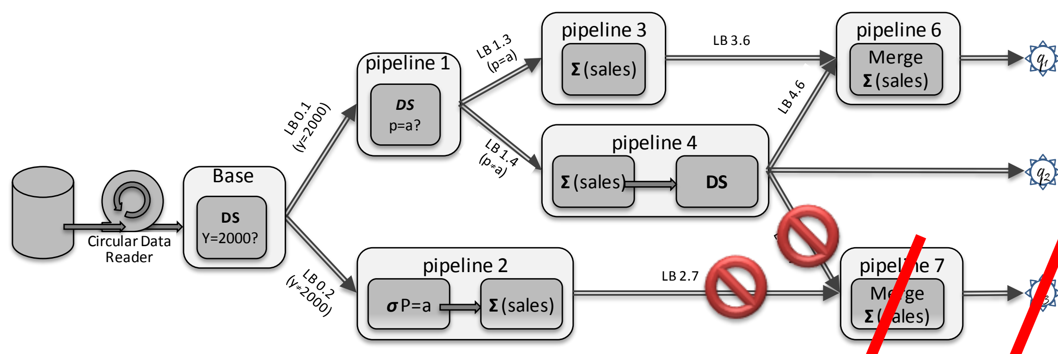


Fig. 6.9 – WPTree reorganization with group removal (1)

When a pipeline is removed, it triggers a reorganization process to update the workload processing tree. The goal is to determine and remove the *WPtree*' pipelines that are producing output results that aren't consumed (i.e. without connected outputs).

Aggregation groups (e.g. SUM ($\sigma_{y=2000,\ p=a}$ (SALES))) that aren't needed by any of the running queries, can be excluded from query processing. As a result, related processing branches (pipeline 2 in the example, illustrated in Fig. 6.10) can also be removed from the workload processing tree (reducing memory usage and data processing).

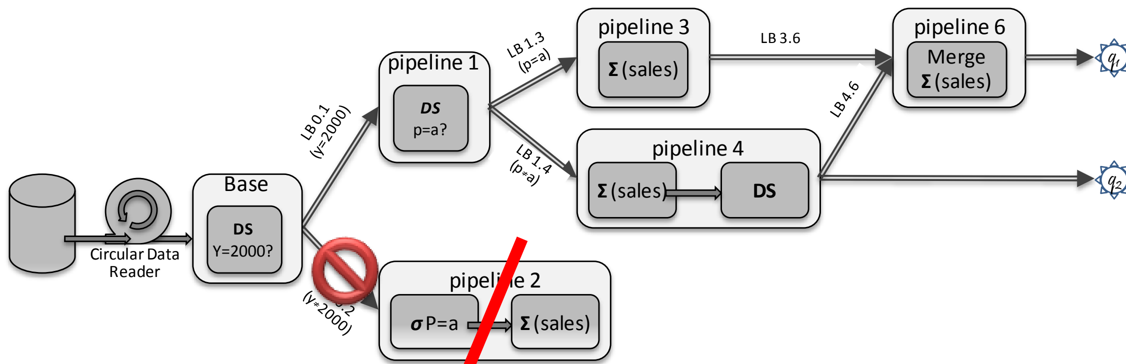


Fig. 6.10 – WPTree reorganization with group removal (2)

The selection predicates of existing branches, pipeline 1 in example, can be pushed forward to upper pipelines, in order to reduce the data volume that has to go through the pipelines for processing. The pushing forward of selection predicates may cause the removal of additional branches, with the associated benefits.

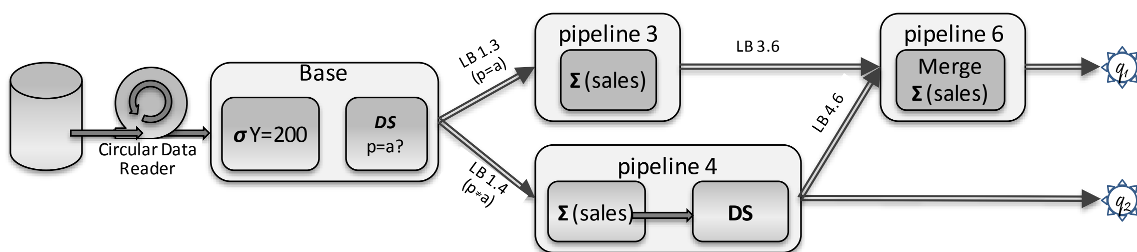


Fig. 6.11 – WPTree reorganization with group removal (3)

As a consequence, when pipeline 2 is removed, the base pipeline switch is replaced with the switch and logical branches in pipeline 1. Then pipeline 1 is also removed. Operators in use by other currently running queries are updated to reflect the removal of query-specific clauses. A data branching reorganization process is triggered to determine if a better logical branching deployment can deliver improved performance. Fig. 6.11 illustrates the layout of *WPtree* after the query removal.

SPIN employs a late garbage collection of disconnected pipelines, allowing that the intermediate results of the disconnect pipelines can be reused by subsequent queries. Intermediate results and corresponding pipelines are discarded when the data that originated the results changes. When needed, a LRU policy of disconnected pipelines is used to free memory.

6.7 Handling data updates and deletes

SPIN processes data in a log-fashion manner, where new inserted data is appended to the existing one. Therefore, deleted tuples are nor physically removed from storage, but only marked as being deleted to avoid being considered from processing. Deleted tuples are only physically removed from storage when a data reorganization occurs.

Whenever a delete statement is issued, a filter selection predicate with the statement predicates is introduced at the beginning of the base pipeline, along with the submission timestamp. For performance purposes, an in-memory bitset (bitmap) marking the "dirty" (deleted) tuples is built and updated as data tuples are evaluated against the delete predicates, as illustrated in Fig. 6.12. These "deleted" query predicates are removed from the base pipeline after a complete circular loop and are then replaced by a bitmap lookup.

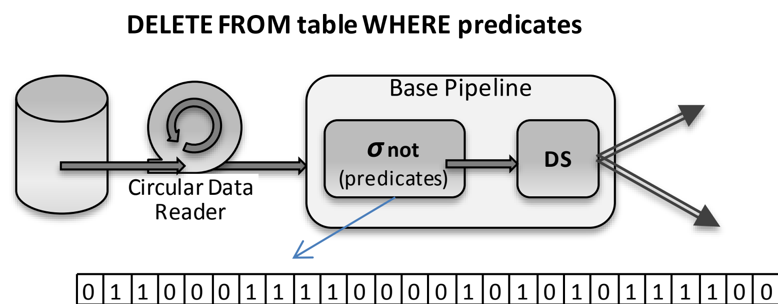


Fig. 6.12 – SPIN - handling deletes

Periodically, or when the data fragments are sparse (with a large number of "deleted" tuples), or when a data reallocation process is executed, the "not deleted" tuples are copied to new data fragments. The old data fragments are removed and the update bitmap is updated accordingly.

SPIN allows data updates to both dimension and fact tables. In practice they are handled in the same manner, since they are stored together into a single relation.

However, we have to distinguish data updates, as correction of previous loaded data, and updates to slowly changing dimensions. In SPIN the later is treated as a simple insert, since the new dimension data express the dimension values that were valid at the time that the events (fact values) occurred. Therefore, changes in dimension that occurred at a given instant are registered when the related fact event (e.g. a sale) occurred. Since data is de-normalized, when fact tuples are loaded they will include the updated versions of referenced dimensions. See Section 4.6 for details regarding data loading and insertion of new data tuples.

For updates of previously loaded data, SPIN treats them as a combination of a delete and an insert, i.e., the updated tuples (new values) are inserted as new tuples and the existing tuples (old values) are invalidated by issuing a corresponding delete statement, as illustrated in Fig. 6.13.

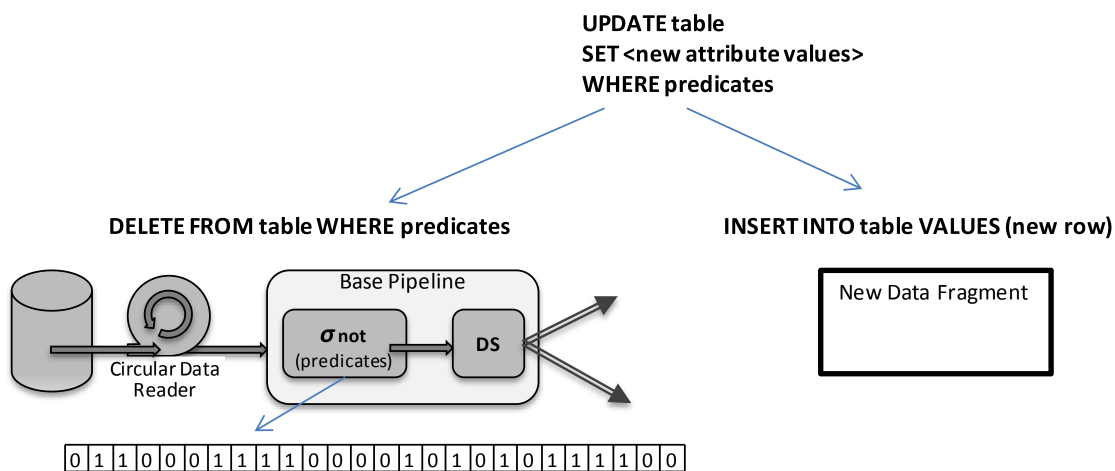


Fig. 6.13 – SPIN - Handling updates

When the percentage of deleted tuples are above a given threshold a coalescing process clear all deleted tuples and frees storage space. See Section 7.3 for more details regarding data fragments partitioning and management.

6.8 SPIN Prototype

We have built a SPIN prototype implemented in Java to evaluate its performance and scalability capabilities. This section presents details of the SPIN prototype, which implements the mechanism discussed above. The prototype was built as a set of flexible and extensible modules, organized in three main layers (illustrated in Fig.6.14): a data access layer, a SPIN core processing layer and a query handler layer.

The SPIN prototype offers a SPIN API for querying and interacting with SPIN. It also offers a JDBC compliant interface that interprets SQL-92 SELECT queries, allowing SPIN to be used as a replacement of the existing DW infrastructure or be used as query accelerator.

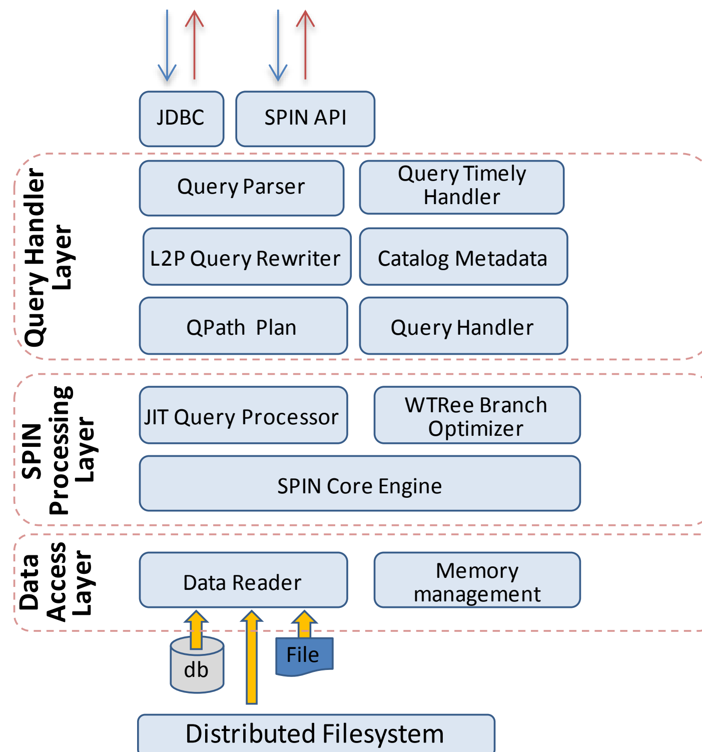


Fig. 6.14 – SPIN prototype diagram - release 1.6.3 (June2013)

The illustrated prototype in Fig. 6.14 relates to the release 1.6.3 (June2013), which has about 32klocs and 153 java classes and do not include several of SPIN optimization mechanisms, such as: data columnar storage organization, compression, partial de-normalized relations with in-memory dimensions, massive data sharding, data loading and snapshot isolation.

6.8.1 The data access layer

The data access layer implements all the functionalities related to gathering the data from storage to the base pipeline for processing. Data is gathered by its main module, the data reader, which can be extended to handle distinct storage locations and physical data organizations. The default data reader accesses tuples physically stored in a row-wise format in a full de-normalized relation as proposed in [Costa, Cecílio, et al. 2011]. We have implemented several data readers, including sequential and mapped memory file using row-wise tuple organization, in-memory object-oriented tuple

organization, compressed row-wise, and SQL based data gathering from common databases using a JDBC driver. The later data reader allows SPIN to act as a middle-layer query accelerator of existing DW infrastructures. This layer also manages a buffering (caching) memory to speedup the data access time, and the processing of frequent accessed data.

6.8.2 The query handler layer

The query handler layer is responsible for handling query requests, for parsing and performing a syntactical and object validation against the information in the metadata catalog. It also contains a module that extracts and validates timely related clauses, to check if they can be satisfied under the current query load. As discussed above, SPIN by default uses a de-normalized ONE data model to avoid joining relations, particularly large dimensions that cannot fit entirely in memory, and therefore providing predictable execution times, but a distinct data reader can be implemented to use others data model organizations. SPIN, in order to be a used as a middle-layer and/or a transparent replacement of the existing DW infrastructure, maintains a representation of the logical star schema model of the DW, regardless of the used data model.

A Logical-to-Physical translator module rewrites the star schema queries according to the SPIN's physical storage model. It provides a JDBC interface to allow seamless integration with users and applications and the submitted SQL queries are syntactically analyzed according to the logical star schema view. As the physical schema may diverge from the logical star schema view, queries are rewritten according to the physical schema representation, into a set of simpler processing tasks with more predictable execution time. A query q, syntactically valid according to the logical schema, is translated (rewritten) into query q_t (or a set of sub-queries) according to the internal physical organization.

Afterwards, a query planner and optimizer builds a Q_{path} execution plan for each rewritten query. A query handler manages the execution and completion of the query Q_{path} execution plan and triggers the *WPtree* reorganization when the query completes (reaches the first logical row).

6.8.3 The SPIN processing layer

The SPIN processing layer handles query execution, maximizing the data and processing sharing among queries. It implements the algorithms discussed in Section 3.1 to plug the specific query pipeline (Q_{path}) to the currently running workload processing tree (*WPtree*). Then, a just-in-time (JIT) query processor uses dynamic coding, to implement all the specific operators, pipelines and data branches required to process the running queries. Afterwards query execution can be started by the SPIN core engine. A *WPtree* branch optimizer is continuously monitoring the execution, the addition and conclusion of running queries. A workload processing tree reorganization is triggered whenever a different deployment can provide higher data and processing sharing and yield better performance.

6.9 Chapter Summary

In this chapter we described SPIN, a data and processing sharing model that delivers predictable execution times to aggregated star-queries even in the presence of large concurrent query loads, without the memory and scalability limitations of existing approaches. We described the mechanisms used by SPIN to embed data and queries into a shared query processing pipeline tree and how SPIN dynamically reorganizes the processing tree. We discuss how data is processed and flows in the workload processing tree (*WPtree*) in order to maximize the sharing of data and processing, and describe how this tree is built and reorganized when new queries are submitted.

Chapter 7

Providing Right-Time Guarantees to Scalable Concurrent Workloads

SPIN enhances the sharing of data and processing and can handle large concurrent workloads. But each query can have a different time target, which can be tighter than the maximum time target (t_{max}) provided by the current *WPtree*. This chapter focuses on how to reorganize the data volume and branch processing of the current deployment, so that it can be able to deliver right-time guarantees.

The chapter starts by proposing a mechanism to determine if the current SPIN workload processing tree can provide right-time guarantees. We proposed a *bitset* processing approach that extends SPIN, which stores the result of predicate evaluation in a bitmap-like structure and replaces costly evaluation with faster lookup operators that use these structures. For large data volumes, where t_{max} is higher than the query time target t_{target}, we propose a parallel SPIN approach, called CARROUSEL, that manages a set of SPIN processing engines in parallel, to speedup query processing and reduce query execution time below required time targets. Tighter right-time guarantees can be provided by extending the parallel infrastructure and redistributing data among processing nodes, but also by redistributing queries, query processing and data branches among nodes, according to their query load and data fragments they store.

7.1 SPIN predicate evaluation overload

In SPIN, any submitted query q has to consider all tuples read from the circular loop, and each tuple that goes through the pipelines has to be evaluated against the

pipelines' predicates. However, some query predicates may be complex and exhibit high computational (CPU) evaluation costs, and thus limit throughput and performance predictability.

The query execution time $\boldsymbol{t_{exec}}$ (q) is constrained by the time required by the Data Reader to read the data ($\boldsymbol{t_{read}}$) and the time needed by the *WPtree* to process that data $\boldsymbol{t_{process}}$, particularly $\boldsymbol{t_{read}}$, since all tuples must be considered for evaluation. While $\boldsymbol{t_{read}}$ is constant and shared among queries, $\boldsymbol{t_{process}}$ is influenced by the computation (e.g. aggregation operators) and evaluation of predicates of each query ($\boldsymbol{t_{eval}}$), and how these can be shared among queries. Since reading and processing is done in parallel, the query execution time is constrained by the largest of these times (max($\boldsymbol{t_{process}}$, $\boldsymbol{t_{read}}$)).

Definition 7.1

Let Q_r be the current query workload, composed by a set of queries, $Q_r = \{q_i\}$, $q_i \in Q$, and let $|Q_r|$ be the number of running queries. Let t_{read} (R) be the time required by the Data Reader to fully read the data of relation R. For a query $q, q \in Q_r$, let $t_{eval}(q)$ be the evaluation time of the predicates of q, and $t_{process}(q)$ be the total processing time of query q (including $t_{eval}(q)$). Query execution time $\boldsymbol{t_{exec}}$ is constrained by max($\boldsymbol{t_{process}}$, $\boldsymbol{t_{read}}$), which we will denote as t_{max}.

For large number of simultaneous queries, the processing time ($\boldsymbol{t_{process}}$) can be larger than the reading time ($\boldsymbol{t_{read}}$), thus endangering the objective of execution time predictability (as illustrated in Fig. 7.1).

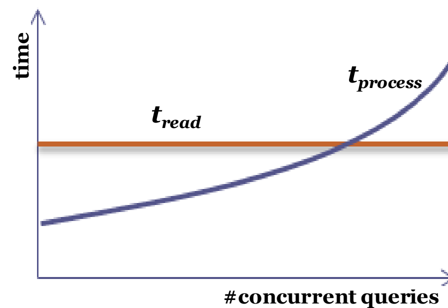


Fig. 7.1 – Increasing processing costs for large WPtree

Even though SPIN shares data reads and processing of common predicates and computational operations, the expectable throughput and predictable performance can be limited by the processing capacity. The submission of new queries that do not fit entirely into the current *WPtree* results in an increase of $\boldsymbol{t_{process}}$, since additional

branches and processing are required. It may also increase t_{read} if the query requires data fragments that were not considered for processing by the current *WPtree*. SPIN throughput and its ability to provide stricter timely targets are constrained by both the data volume and the complexity of the *WPtree*.

For large query loads, resulting in a complex workload processing tree, a query *q* with a given time target t_{target} below t_{read} may not be timely processed because of the complexity and costs associated to the evaluation of branch predicates.

When the current deployment cannot provide the required timely targets for some queries, either due to the reading costs or the processing costs, there is the need to optimize the fragment processing within a node or to use parallel processing.

7.2 The bitset branch processing approach

The evaluation time (t_{eval}) is constrained by the number of predicates and branches of the current *WPtree*. Queries with common predicates are combined in common branches, in order to avoid duplicate evaluation of the same predicates. A submitted query has to process and evaluate each of its predicates, even though similar queries, with common predicates, have been previously processed in the past. As predicates of common queries are associated to common branches, a branch with a given predicate can be repeatedly built or may persist over time while at least one running query uses it.

In SPIN, each branch is associated with a predicate, or set of predicates, which is evaluated against every tuple that flows along the branch. Since tuples flow using the same reading order, if data doesn't change, the evaluation of the branch predicates against every tuple that flows along the branch will not change. The result of predicate evaluation will be the same as the last time it was evaluated.

To avoid subsequent evaluation of unchanged data tuples, we extended SPIN with a *bitset* processing approach, which maintains the boolean evaluation of selection operators as tuples flow through data branches. We build a branch *bitset* (bitmap) according to the branch' predicates, where each bit represents the boolean result of the predicate evaluation (true/false) applied to a corresponding tuple index.

Fig. 7.2 – An example of a *WPtree*

In the example illustrated in Fig. 6.7, the aggregation query q_1 = SUM ($\sigma_{y=2000, p=a}$ (SALES)), will produce two bitmaps, one for predicate `y=2000` and another for `p=a`, as illustrated in Fig. 7.3. Subsequent queries that have any of these predicates can replace the predicate evaluation with a lookup operator to the corresponding bitmap.

tuples	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
	2000	2001	2000	2000	2001	2000	2001	1999	2000	1999	2001	1999	1999	2001	2000	1999	1999	1999
	c	a	a	a	a	a	d	b	c	a	c	a	a	d	c	c	c	c
	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
y= 2000	1	0	1	1	0	1	0	0	1	0	0	0	0	0	1	0	0	0
p= a	0	1	1	1	1	1	0	0	0	1	0	1	1	0	0	0	0	0

tuples	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
	1999	1999	1999	2001	2001	2000	1999	2000	2000	1999	2000	1999	2000	2000	2001	2000	1999	1999
	a	b	b	a	b	c	d	d	a	c	a	c	c	d	d	c	d	a
	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
y= 2000	0	0	0	0	0	1	0	1	1	0	1	0	1	1	0	1	0	0
p= a	1	0	0	1	0	0	0	0	1	0	1	0	0	0	0	0	0	1

Fig. 7.3 – Online predicate bitmap indexes

The query q_2 = SUM($\sigma_{y=2000, p=b}$ (SALES)) can be rewritten as SUM ($\sigma_{bitmap_lookup, p=b}$ (SALES)). As a consequence, the query specific pipeline (pipeline 4 in Fig. 6.7, which consumes data tuples from pipeline 1), can then be directly connected to the base pipeline Data Switch (DS). This DS switches relevant data tuples to pipeline 4 according to the bit value at corresponding bitmap position.

As SPIN processes tuples in a circular fashion, when the relation reaches the end, it restarts reading from the beginning. Therefore, future evaluations of a tuple can take advantage of the existence of this *bitset*, since the selection operator that evaluates the predicate can be replaced by a fast lookup operator that lookup the corresponding position in the *bitset* to gathers the result. *Bitsets* are small and reside in memory in order to avoid introducing overhead at IO level. This is particularly relevant for predicates with high evaluations costs.

The following sections discuss how the *bitsets* are created, the *bitset* operators and the changes in the workload processing tree.

7.2.1 Creation of Bitsets

A *bitset* is built on-the-fly, as tuples go through the branches and are evaluated, with minimum overhead, since these results are stored in an in-memory data structure. Fig. 7.4 illustrates the process of building the *bitset* for each branch.

In the figure, the reader scans the table data tuples (rows with the values of two attributes, year and product type, are illustrated in the figure). Tuples read are placed in an initial pipeline (leftmost *DS* in the figure) and then consumed by the connected branches. A branch evaluates each tuple and stores the result of the predicate evaluation in the *bitset* (the *bitset* is illustrated in the figure as a set of bits depicted on top of each branch). The *bitset* stores 1 or 0 in each position, meaning that the row at the corresponding position evaluates to true or false, respectively. This is very important, since it makes it possible in every future evaluation of each row, to decide whether that row should proceed to the next step in the branch, or not, based on the simple lookup of the *bitset*. This avoids most of the evaluation costs.

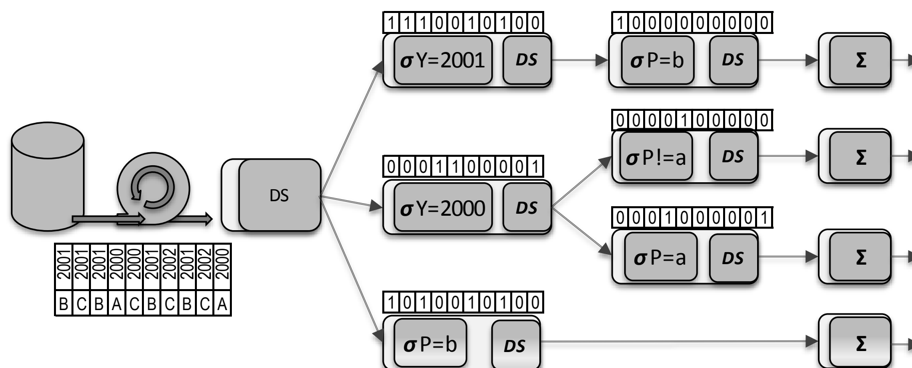


Fig. 7.4 – Predicate evaluation and building the bitset

For a given attribute, a *bitset* can be built for each value of the attribute domain (e.g. 11, 12, 13 …), or for sets of values, or intervals, of the attribute domain (e.g. [10;13]). In order to avoid introducing additional overhead, *bitsets* are built according to the selection predicates of the queries that are being submitted. When a new selection operator is deployed in the ***WPtree*** that does not have a matching *bitset*, it starts building a new *bitset* with the result of the selection predicate.

This *bitset* is kept in memory and shared by all branches that can use it, allowing future evaluations of these tuples to be replaced by a fast *bitset* lookup operator.

7.2.2 Bitset lookup operators

Branches can be built, or reorganized, to use *bitsets* to evaluate tuples. A selection operator (***σ***) with a matching *bitset* is replaced with a special bit-selection operator (***σbit***), which perform bit lookups to the corresponding index position in the *bitset,* thus avoiding the evaluation of these tuples. Fig. 7.5 depicts an equivalent workload processing tree where the selection operators ***(σ)*** are replaced with ***σbit*** operators.

A branch with a selection operator (***σ***) that perfectly matches a *bitset* is replaced with a ***σbit***. Selection operators that do not have a matching *bitset* can also take advantage of *bitset* evaluation by combining the existing *bitsets* for other values of the attribute domain. For that purpose, there is also a special not bit-selection operator (***σ!bit***) that evaluate as true all the index positions in the *bitset* that are 0. ***σ!bit*** is equivalent to NOT ***σbit.***

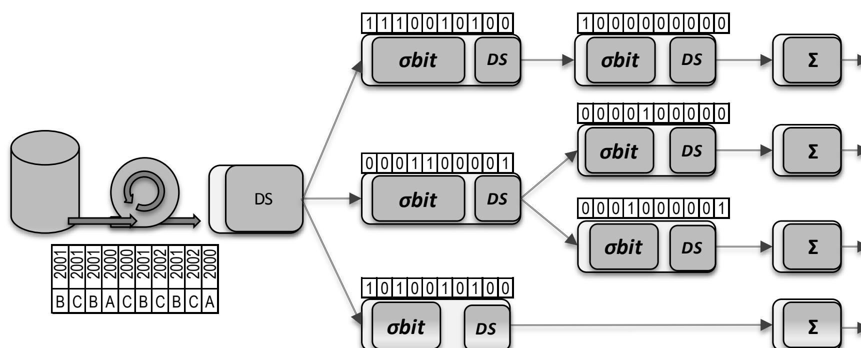


Fig. 7.5 – A branch processing layout using bitsets

For instance, consider that the domain of the attribute Y (year) is 2000 and 2001, `D(Y)={2000,2001}`. If there is a *bitset* for 2000, ***bitset*(y=2000)**, the selection operators ***σ*** **(y=2001)** can be replaced with a ***σ!bit*** **(y=2000)** operator that applies a bitwise **NOT** to ***bitset*(y=2000)**. The result is equivalent to **NOT(*σbit* (y=2000)).** Fig. 7.5 shows how *bitset* processing can be employed to boost performance even when a perfectly matching *bitset* does not exist. Fig. 7.6-a) depicts the domain complement.

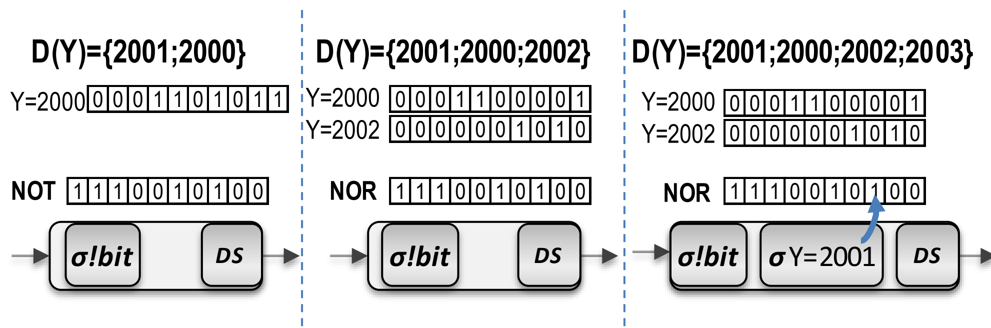


Fig. 7.6 – Selection operators a) NOT b) NOR c) NOR

Fig. 7.6-b) depicts the scenario of a wider domain, where there exists a *bitset* for each of the values of the attribute domain except for the value Y=2001. In this case, the ***σ!bit*** operator does a NOT to the OR of the *bitsets*, **NOT(OR(*bit* (y=2000),*bit* (y=2002)))**, or simply **NOR(*bit* (y=2000),*bit* (y=2002)).** The domain complement (Fig. 7.6-a)) is a particular case, where the domain has only two distinct values.

Bitset processing can still be used when the number of values, of the attribute domain, without a matching *bitset* is greater than 1 (Fig. 7.6-c). In this case, the selection operator ***σ*** is maintained in the branch, but it is preceded by a ***σ!bit*** that applies a **NOR** to the existing *bitsets*, thus obtaining a *bitset* with all the index positions that are certainly false, marked as 0. The goal of this ***σ!bit*** is to avoid the ***σ*** operator from evaluating these tuples that are known to be false. The remaining index positions, which can be evaluated as true or false, are set with all 1'. As the ***σ*** operator evaluates these remaining tuples, it updates and completes the *bitset* with the result of the evaluation (illustrated in the figure with a blue arrow).

7.2.3 Mixed branch processing: branches with and without bitsets

At any given time, SPIN may have branches with *bitset* operators and other branches that have to evaluate the predicates, because there isn't a *bitset* that matches the selection predicate. Branches that use *bitsets* are pushed forward and connected directly to the base data pipeline to maximize the sharing costs, and to reduce the overall number of tuples that have to be evaluated with selection operators (***σ***).

Newly created branches that do not have ***σbit*** operators are connected, as usual, to the base pipeline, or to other branches with common predicates, regardless if they have a *bitset* or not. Whenever a branch ends building a *bitset*, it is replaced by an equivalent branch that performs the same evaluation through *bitset* operators (***σbit***).

Since *bitset* processing is faster than the tuple predicate evaluation, branches that contain *bitset* operators are reorganized and pushed forward to the base pipeline.

To create a new *bitset*, a branch can use existing *bitsets*, to narrow down the tuples that have to be evaluated against the branch predicate. When a new branch is added to the *WPtree*, the corresponding selection predicate is associated with a *bitset* composed with all 1's. If exists a *bitset*, or a set of *bitsets*, related to the branch' predicate, then the *bitset* of this new branch is updated accordingly.

Over time, predicates more frequently used will have a corresponding *bitset*, and therefore will deliver faster query processing times.

7.2.4 Merging bitsets along the query logical path

For a query to be processed, tuples have to follow its logical data path, which is composed by a set of branches organized in order to maximize the sharing and processing costs among queries with common predicates. *Bitset* operators (***σbit***) allow faster evaluation times. However, the execution time of query can be even further improved by replacing the branches, which belong to its logical path, that have ***σbit*** operators, with a single branch with a ***σbit*** operator that evaluates the *bitset* that corresponds to the bitwise **AND** of branches *bitsets*.

When in a logical data path, two or more branches use exclusively *bitsets* to evaluate tuples, then these branches are merged into the last one, with a *bitset* that is a logical AND of these branches. The resulting branch replaces the merged branches, or is directly connected to the base pipeline. The main goal is to filter as soon as possible the data tuples that are relevant to a query, before reaching the branches that evaluate tuples using selection operators.

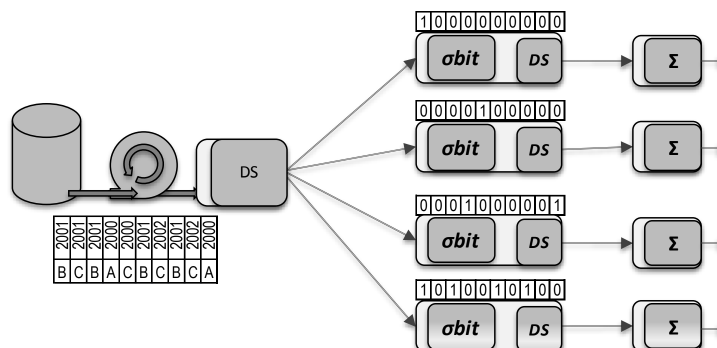


Fig. 7.7 – Branch processing with a bitwise AND of the logical path branches

Fig. 7.7 depicts the new deployment, where the query logical data paths are built with less data branches, where the last branch of each query path in the previous deployment (Fig. 7.5) is substituted with a branch that evaluates tuples using single *bitset* that represents the logical evaluation of all the *bitsets* of the logical data path.

In the figure, the *bitset* of query 1 (the topmost branch, which has only the first bit set to 1) was built by the selection operator (***σ***) of the last branch of the logical path, but it could also be built by applying a AND to *bitset*(*y=2000*) and *bitset*(*p=b*). The four queries depicted in the figure are evaluated with dedicated branches, each with a single ***σbit*** operator. These *bitsets* can be pushed forward to the base pipeline and be used by the data reader to reduce the cost of getting and forwarding the data.

7.2.5 Pushing forward bitsets to the data reader

If all the branches connected to the base pipeline use ***σbit*** operators to filter the tuples that relevant for each branch, then it is possible to build a composed *bitset* with all the index positions that are relevant for at least one of the branches. This *bitset*, named the *WPbitset*, indicates the relevant tuples required by current workload, and can be used to control the data to gather from storage and optimize the IO reading cost. The *WPbitset* is created by applying a logical OR to all the branches' *bitsets*.

In a mixed environment, with some branches using ***σbit*** operators and others not, an all 1's *bitset* is considered in the bitwise OR, when exists at least one first level branch that does not use a ***σbit*** operator to evaluate tuples. For the previous deployment, the result of the merging OR is depicted in Fig. 7.8.

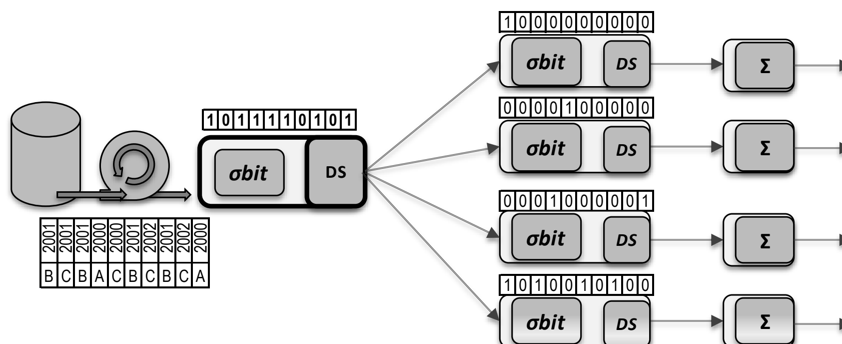


Fig. 7.8 – Data Reader Bitset computed as a bitwise OR of the branches bitsets

SPIN partitions data in data chunks. *WPbitset* allows the Data Reader to read from storage only the chunks of data that are relevant for the workload processing tree, and thus reducing the execution time of the currently running queries. Even when the

data reader cannot use *WPbitset* to filter data chunks, for instance when at least one tuple of each chunk is required, and thus all chunks have to be read, it can use it to decide which tuples to place in the pipeline (only those tuples pinpointed by the *WPbitset*).

Queries with logical data path containing ***σbit*** operators, do not have to wait for the last tuple (logical end tuple) to end execution and return the result. As soon as the number of processed tuples, tuples that satisfied the *bitset* reaches the *bitset count* (hamming distance), then the query can stop execution since all the tuples relevant for the query (which satisfies the selection predicates) was already processed.

When that occurs, the query execution can fall below the barrier imposed by the IO cost of reading the full relation (in a circular loop), since with *bitsets* a query can end whenever all the set positions in the *bitset* as been processed.

7.2.6 In-Memory Bitmap Management and Retention

Bitsets for new predicates are built on-the-fly according to the query pattern, but since the physical memory to hold *Bitsets* is limited, there is the need to recycle (drop) some of the *bitsets*. SPIN implements a *Bitset* replacement policy, where *Bitsets* are built, merged and dropped taking in consideration the following factors:

- ***RecentlyUsed*** (`LRU`) – *bitsets* used more recently used should be maintained, since there is a high probability that it will be used again in the near future.
- ***HitRatio*** (`hitratio`) – *bitsets* used more frequently used should be maintained to provide improved performance.
- ***Rebuildability*** (`rebuild`) – ability to rebuild a *bitset* by applying bitwise operations to existing *bitsets*. For instance the *bitset* for predicates (`P=a AND Y=2000`).

y=2000 AND p=a	0	0	0	0	0	1	1	1	0	0	0	1	0	0	0	0	0	1	1	0	0	0	0	1	0	0

  is a candidate for removal if it can be rebuild with a bitwise `AND` between

y=2000	0	0	0	1	0	1	1	1	1	1	1	1	0	1	0	0	0	1	1	1	0	0	0	1	0	0
p=a	0	0	1	0	1	1	1	1	0	0	0	1	0	0	1	1	1	1	1	0	0	0	1	1	1	1

- ***Predicate evaluation cost*** (`cost`) – *Bitsets* of more costly predicates should be maintained, since it yields higher throughput and less computation time.

Numerical comparisons are faster than character comparisons and other comparisons that include complex operations.

- ***Hamming Weight (population count)*** (`count`) – number of bits set as 1. A *bitset* with a higher population count is a candidate for removal, meaning that there is a large number of tuples that satisfy the predicate. Therefore it only avoids that a small fraction of tuples are not read from storage.
- ***Bitset Size*** (`size`) – the *bitset* size (in kb). All *bitsets* have the same size, unless some sort of compression is employed.

Recall that any removed *bitset* can be completely rebuilt after a complete circular loop, and therefore, subsequent queries can then take advantage of it to deliver improved processing time. However since they are built on-the-fly according to query predicates, the number of *bitsets* may rapidly exhaust the memory size *Msize* allocated for *bitsets*.

By default, a Least Recently Used policy is used, which returns a list of *bitsets* (β_{LRU}) order by the last time a *bitset* was used (less recently used first). While the memory required to hold all the *bitsets* is still above *Msize, bitsets* are removed sequentially according to the given list (by default β_{LRU}).

However, other policies can yield better results, for instance a less frequently used *bitset* may be used to avoid high predicate evaluation costs, and therefore for improved performance it should not be removed. For that reason, for each of the above factors, we implemented a set of methods that return lists of *bitsets* ordered according to the factor. Considering β as the set of existing *bitsets*, let

$\boldsymbol{\beta}_{\boldsymbol{LRU}} \leftarrow$ *sortByLessRecentlyUsed* (β) - be the list of *bitsets* ordered by the last time they were used (less recently used first)

$\boldsymbol{\beta}_{\boldsymbol{hitratio}} \leftarrow$ *sortByLessUsed*(β) - be the list of *bitsets* ordered by their usage, (less frequently used first)

$\boldsymbol{\beta}_{\boldsymbol{rebuild}} \leftarrow$ *sortByRebuildability* (β) - be the list of *bitsets*, in an ascending order according to the number of *bitsets* that it needs to be rebuilt. *Bitsets* that cannot be rebuilt are set in the end of the list.

$\beta_{cost} \leftarrow sortByPredicateCost(\beta)$ - be the list of *bitsets* in an ascending order according to the cost of evaluating the predicate without the *bitset* (lowest costs first). The cost is collected when the *bitset* is built.

$\beta_{count} \leftarrow sortByHammingWeight(\beta)$ - be the list of *bitsets* in a descending order according to the population count (higher counts first).

$\beta_{size} \leftarrow sortBySize(\beta)$ - be the list of *bitsets* in a descending order according to the memory size that it occupies (greatest first).

When a new *bitset* is being created and there isn't enough space to holding it, the *bitset* is included in the list of considered *bitsets* (β). We also offer a method *sortByWeight,* which determines an ordered list of bitsets (β_{weight}), based on a mixed of the above methods, through the definition of a set of weights. The function used to order this list is determined by the sum of the weight multiplied by the index position in the corresponding list,

$$f_{weight} = W_{LRU} \times \beta_{LRU} + W_{hitratio} \times \beta_{hitratio} + W_{rebuild} \times \beta_{rebuild} + W_{cost} \times \beta_{cost} + W_{count} \times \beta_{count} + W_{size} \times \beta_{size}$$

$\beta_{weight} \leftarrow sortByWeight(\beta)$ - be the list of *bitsets* in a ascending order according to the function f_{weight} presented (lowest first). It the weights are not specified, it assumes by default that all weights are set as 0, except for W_{LRU} which is set to 1 (the default LRU policy).

7.3 Optimizing the processing of data fragments

After a *bitset* is created, we use it to create a fragment level *bitsets* (*fbitset*), where each bit represents a data fragment *f*, to determine which data fragments are required by each predicate. A *fbitset* is significantly smaller and is used to optimize the query execution time by reducing its lifetime or postponing the beginning of its execution. It can also be used to early end the query execution, before completion of the circular loop, when all the relevant data fragments have already been processed. Fig. 7.9 illustrates this concept.

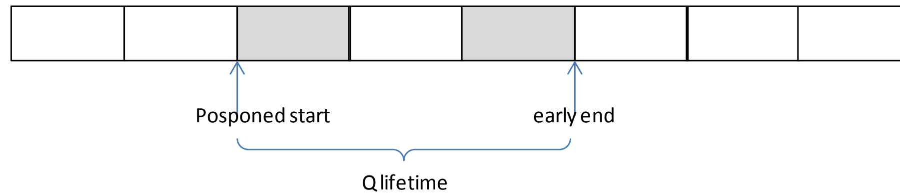


Fig. 7.9 – Early-end execution and query lifetime

When a query q is submitted, a fragment level *bitset* is created for the query q, $fbitset_q$, with all the bits set to 1 (meaning that data fragments have to be considered for execution). Afterwards, SPIN analyzes the query predicates to determine if any of the existing *fbitsets* match the query predicates. The query $fbitset_q$ is then updated accordingly to the *fbitset*s that match the predicates of q. The goal is to determine the data fragments that do not have relevant data for the query (based on the predicate *fbitsets*), and exclude them for execution. The Data Reader may even skip the data fragments that are not required by the current workload processing tree.

Definition 7.2

Let f be a fragment, F be a set of fragments, $F=\{f_i\}_{i=1,\ldots,|F|}$ *, |F| is the number of fragments of F, and Fbitset be a set of fragment level fbitsets,* $Fbitset=\{fbitset_j\}$*. For a query q,* $q \in Q$*, let* $fbitset_q$ *be the query fragment level bitset. For all selection predicate σ of q with a matching fragment* $fbitset_\sigma \in Fbitset$*, then* $fbitset_q)$ *is updated accordingly with* $fbitset_\sigma$*. The set of fragments that q has to process,* F_q*,* $F_q \subseteq$*, can be determined as* $F_q = \{f_i : \forall\, fi \in F \wedge fbitset_q\,(i) = 1\}$.

Every time a query q processes a data fragment i, the corresponding index in the bitset is set to 0 ($fbitset_q(i)=0$). The query can end its execution as soon as the hamming weight of $fbitset_q$ is 0, i.e., $fbitset_q$ is set to all 0.

The Data Reader determines which data fragments are required by the current query load Q_r, and have to be read from storage, by performing a bitwise OR to every $fbitset_q$ of the current query load, i.e., $fbitset_{Qr}=OR\{\ fbitset_q\ \}_{,q\in Qr}$. With this $fbitset_{Qr}$ we determine the set of fragments F_{Qr}, $F_{Qr} \subseteq$ F, that have to be read from storage. The remaining data fragments, $F \setminus F_{Qr}$, may be skipped to reduce the overall IO cost, the major factor in query performance, as illustrated in Fig. 7.10.

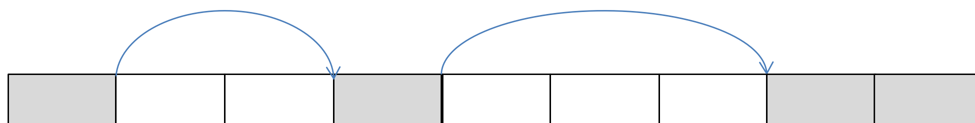


Fig. 7.10 – Data reader fragment skipping

While there is a performance boost at IO level, fragment skipping is also interesting and relevant for pipeline processing. Pipelines with predicates that match existing *fbitsets* can use them to stall processing, for fragments that are market as 0, and thus releasing processing resources for other pipelines.

For every query q: $q \in Q_r$, with a corresponding $fbitset_q$, we can control its execution time by managing (changing) the order that data fragments are read and processed.

Sequential Fragment reading order

1	2	3	4	5	6	7	8	9	10	11	12	13

Changed fragment reading order

1	**7**	3	4	**9**	**10**	**2**	**5**	**6**	**8**	11	12	13

Since the size of data fragments are large enough, changing the order that they are read does not introduce significant IO overhead (since we are introducing more disk seeks). This is particularly interesting to guarantee time targets and for providing execution times smaller than the cost of reading all the data fragments F.

Definition 7.3

For a set of fragments F, $F=\{f_i\}_{,i=1,\ldots,|F|}$, let $S_F=\{s_1,..,s_{|F|}\}$ be the sequence that fragments will be processed, such that $\forall s_i, s_j \in S_F, {}_{i<j}$: $s_i=f_g$, $s_j=f_h$: $f_g, f_h \in F$, $f_g \neq f_h$. For a query q, $q \in Q$, let F_q be the set of fragments that q has to process, $F_q \subseteq F$, and let F_{Qr} be the set of fragments that the currently running queries Q_r have to consider for processing, $F_{Qr} \subseteq F$, such that $F_{Qr} = \bigcup F_{p:\forall p \in Qr}$.

We denote S_{Qr} as the minimum sub-sequence of S_F that contains all fragments of F_{Qr}, such that $S_{Qr} \subseteq S_F$, $\forall f \in F_{Qr}$: $f \in S_{Qr}$, and denote S_q as the minimum sequence of S_{Qr} that contains all fragments of F_q, $S_q \subseteq S_{Qr}$, $\forall f \in F_{q:}$ $f \in S_q$. We define the lifetime of q as $|S_q|$.

When a query q is submitted, the processing infrastructure has to assess if is able to process the query within the time target. It determines which data fragments are required by query (F_q) and by the current query load (F_{Qr}), and then estimates the time needed to process F_q and F_{Qr}. It is important to notice that after a fragment f has been processed, F_q and F_{Qr} are updated as $F_q = F_q \setminus \{f\}$ and $F_{Qr} = F_{Qr} \setminus \{f\}$. The t_{target} of each of

the running queries is updated accordingly with the elapsed time $\forall\ q \in Q_r,\ t_{target}(q) = t_{target}(q) - t_{process}(f)$.

Definition 7.4

For a query q, $q \in Q$, let $t_{target}(q)$ be the time target of q, and $t_{process}(f)$ be the time required to read and process a fragment f, $f \in F$ and $t_{process}(F)$ be the time required to process the fragments F, such that $t_{process}(F) = \sum t_{process}(f_i)$, $_{i=1,\ldots,|F|}$.

If $t_{process}(F_{Qr} \cup F_q) < t_{target}(q)$ then q can be timely processed

if $t_{process}(F_q) > t_{target}(q)$ then q can be timely processed iff

$$\exists\ S_{Qr}: t_{process}(S'_{Qr}) < t_{target}(q) \wedge t_{process}(S'_{Qr}) < t_{target}(Q_r)$$

otherwise q cannot be timely processed by the processing infrastructure

When $t_{process}(F_q) > t_{target}(q)$, q can still be timely executed if we can determine a different sequence order S'_{Qr} for the fragments that have to be read and processed, that guarantee the timely execution of q and Q_r.

Definition 7.5

For a sequence of fragments S, let S(i) denote a fragment of S at index i. Let p(f,S) denote the index position where f is within S, such that $\forall f \in F, \exists j: p(f,S)=j \wedge S(j)=f$. For a query q, $q \in Q$, let S_q be the sequence of fragments that q has to process, q can be postponed iff $f = S_q(1) \wedge p(f, S_{Qr}) > 1$, i.e. $S_q(1) \neq S_{Qr}(1)$.

The minimum lifetime of q can be determined as minimum sequence of data fragments of Q_r, S_{Qr} such that

$$\forall f \in F_q, f \in S_{Qr} \wedge \forall g \in S_{Qr}, f \neq g: |S'_{Qr}| > |S_{Qr}|,\ S'_{Qr} = S_{Qr\ f \leftrightarrow g}, \wedge t_{process}(S'_{Qr}) < t_{target}(Q_r)$$

where S'_{Qr} is equivalent to S_{Qr} but with the position of f exchanged with g.

A query q, $q \in Q$, with F_q as the set of fragments that q has to process, can be early ended, i.e. can end its processing before the full circular loop is completed, when $F_q \subseteq F$ and $|S_q| = 0$ (or $S_q = \emptyset$). Remember that as data fragments are being processed, F_q and S_q are updated accordingly.

When for a query q, $t_{process}(F_q) > t_{target}(q)$ and $\forall S_{Qr}$: $t_{process}(S_{Qr}) > t_{target}(q) \vee t_{process}(S_{Qr}) < t_{target}(Q_r)$, the query q cannot be timely executed unless the data processing infrastructure is rearranged.

7.4 Parallel Processing - CARROUSEL

CARROUSEL is a flexible fragment processor that uses idle nodes, or nodes currently running less time-strict queries, to process some of the fragments required by time-stricter queries, on behalf of the fragment node's owner, as illustrated in Fig. 7.11. By reducing the data volume to be processed by a node, it can provide faster execution times. Alternatively, it may reduce query processing by distributing some logical data branches among nodes with replicated fragments.

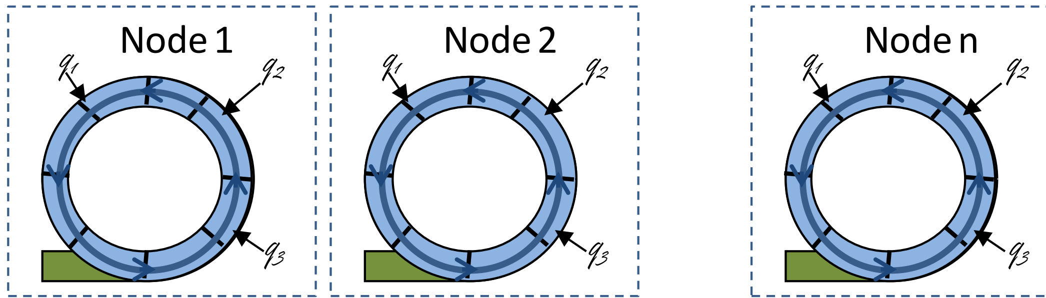


Fig. 7.11 – Carrousel (parallel processing)

Recall that query execution time of a query $q, q \epsilon Q$, is $t_{exec} = \max(t_{read}, t_{process})$ over all data tuples, regardless of the query load. When the time a data reader dr needs to read all tuples from storage t_{read}, is greater than the query time target t_{target}, then the issue is related to IO reading costs. Therefore it is necessary to increase the IO throughput, with additional Data Readers, either locally using different storage disks or by using a set of parallel nodes.

To overcome such issue, a number n_{dr} of concurrent data readers dr_i, $_{i=1,\ldots,ndr}$ may be set on opposite positions of the circular loop, at equal-size distances, $dist(dr_i, dr_j) = |F|/n_{dr}, \forall dr_i, dr_j,\ _{i,j \in \{1,\ldots,ndr\}}$. As a consequence, the relation O_d is fully read by the set of data readers dr_i, after each dr_i has read $|O_d|/\ n_{dr}$ tuples.

The reading time t_{read} is then reduced to $t_{read}\ /n_{dr}$, but only if every data reader dr_i, can perform sequential reads of O_d without being interrupted. It assumes that each data reader dr_i is an independent process with a dedicated disk and bus, or is running in a different node.

Fig. 7.12 illustrates the carrousel with three Data Readers orchestrated with two distinct approaches, where the Data Readers are positioned in equal-size distances.

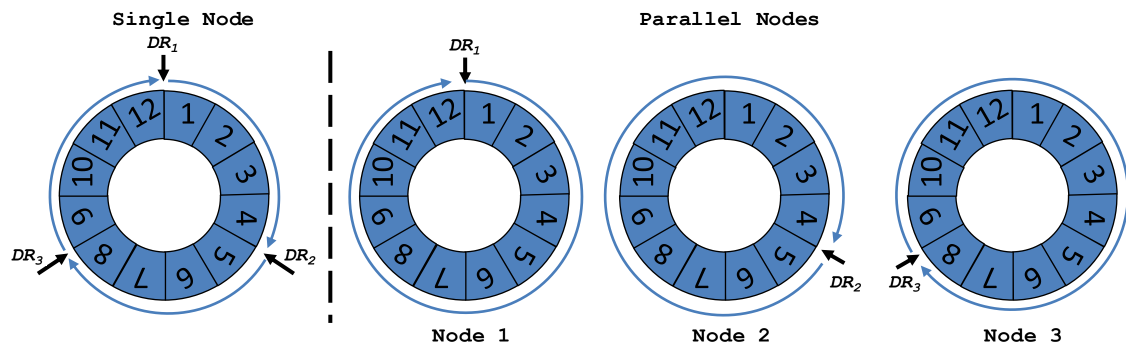


Fig. 7.12 – Data readers at different positions a) single node b) multiple nodes

In the figure, the data is replicated in each node and the data readers in the nodes are lopping around all fragments the circular loop. While this is the most flexible approach for balancing data and query load, it requires that all the data fragments be replicated to each node and therefore may require more network bandwidth.

7.4.1 Managing fragment metadata

While the use of multiple data readers in a node is straightforward, since it just increases the throughput at which tuples are read and placed into the base pipeline, the parallel processing approach requires a control of which data fragments exists in each node, which has been processed and which are missing for each of the running queries. For simplicity reasons, from now on, we use the term node to express an independent process with a dedicated disk, bus, processing unit and memory, which is running in a different physical node, or along with other processes.

Data is partitioned in data fragments (or chunks) and distributed among available nodes. A fragment can be replicated in several nodes. Data is partitioned in fragments which are included in the circular loop for processing. The execution of a query ends when all tuples stored within data fragments are processed and the circular logical loop is completed. SPIN uses a set of fragment-level bitmaps to enhance the management and processing of the data fragments. In particular the following

- ***allFragmentsBitmap*** –a bitmap representing all the data fragments that make part of the relation. Each bit represents a specific data fragment. In a parallel deployment, each node will have a replica of this bitmap, and a change in the bitmap is replicated to all nodes.

- ***availableFragmentsBitmap*** – a bitmap, similar to the allFragmentsBitmap, representing all the data fragments that are currently available in a node. A node could hold several data fragments, but some may be active (which must be included in local circular for processing) or be inactive (e.g. a representing a replica of data fragment that is inactive in a node but it is active in another node, and therefore should not be include in the node local circular loop). In parallel deployments, each node will have a local *availableFragmentsBitmap* representing the data fragments that the node has available for processing.
- ***activeFragmentsBitmap*** – a bitmap, similar to the availableFragmentsBitmap, representing all the available data fragments that are active and therefore have to included in the circular loop for reading. Data fragments that are unavailable (e.g. under maintenance, or been replaced by other) are market as inactive (bit set as '0') and will not be considered for processing. In parallel deployment, each node will have a local *activeFragementsBitmap* representing the data fragments that the node has active and must consider for processing.
- ***updatedFragmentBitmap*** – a bitmap representing all the data fragments that were or are being updated. In a parallel deployment, when a data fragment is being updated in a node, the remaining nodes have to mark the corresponding data fragment as dirty. Only one can node can mark a data fragment as updated.
- ***dirtyFragmentBitmap*** – a bitmap representing all the data fragments that are dirty (have been updated or are under update by other nodes) and that must be refreshed.

For consistency purposes, for updated data fragments, we also store a timestamp corresponding with the point in time where the changes in the data fragment have committed. Similar timestamp is also stored for dirty data fragments. The goal of this timestamp is to guarantees that the results of queries submitted after this timestamp are computed over the updated data fragments.

Whenever a new node is added to the processing infrastructure, these bitmap are created in the node and updated as fragments are replicated and become available in the node. Fig. 7.13 illustrates a setup with 3 processing nodes, where the active fragments in each node are depicted in blue, and the available fragments are those that are active and also the fragments depicted in grey.

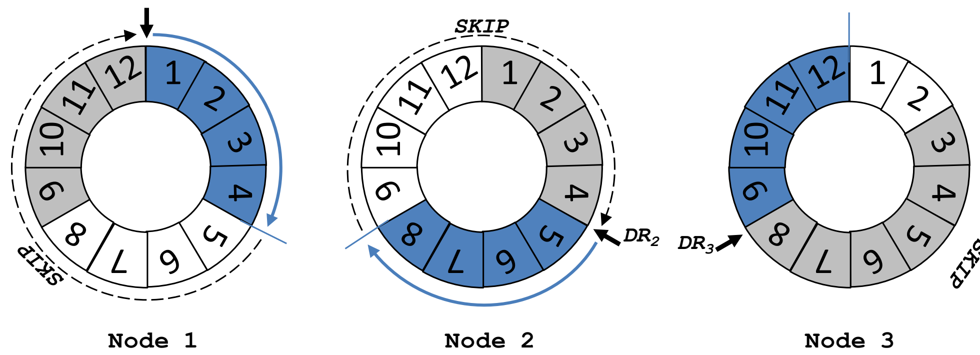


Fig. 7.13 – A parallel deployment with inactive fragments in each nodes that can be skipped

CARROUSEL includes a management module, which manages the information regarding the nodes that are active at a given moment, and its current performance indicators.

For each node, and regarding the data fragments, it also maintains information regarding which fragments are allocated, which are active and which are required by the workload processing tree that is currently running at the node. The fragment that is currently being processed and the sequence order of the following fragments are also maintained in order to be able to optimize and guarantee timely execution. To optimize the data redistribution, data fragments may be split and moved among nodes, and therefore such information is also maintained.

The fragment level and the node level metadata information are used by carrousel to maximize, when the rebalancing is required, the clustering of data and branch processing in the same node.

7.4.2 Balancing Data and Query processing among nodes

When multiple nodes are needed to provide right time guarantees, CARROUSEL distributes query and data processing among nodes according to the reason why the query target is not meet.

Definition 7.6

Let q be a new submitted query, $q \in Q$, added to the set of running queries Q_r, and let $WPtree_r$ be the workload processing tree of Q_r. Let F_q be the set of fragments that q has to process, t_{target} as the time target of query q, $t_{read}(F_q)$ the time needed to read F_q and $t_{process}(F_q)$ the time required to process F_q. Data and processing has to rebalancing, according to the following conditions:

If $t_{read}(F_q) > t_{target}(q)$*, then data load has to be rebalanced*

If $t_{process}(F_q) > t_{target}(q) \wedge t_{process}(F_q) \gg t_{read}(F_q)$ *then processing has to be rebalanced*

If $t_{process}(F_q) > t_{target}(q) \wedge t_{process}(F_q) > t_{read}(F_q)$ *then data and processing issue*

When there is a data reading issue, $t_{read}(F_q) > t_{target}(q)$, data have to be rebalanced among other processing nodes, in order to reduce the data volume that a node as to read, until it is below the time target $t_{target}(q)$. Alternatively, some queries have to be deliberately not timely executed, particularly those with tighter time targets or having requiring more data fragments.

When there is a processing issue, $t_{process}(F_q) > t_{target}(q)$, and the system is unable to timely handle the *WPtree* complexity, we could rebalancing the processing among nodes, by splitting the *WPtree* and allocating some branches other nodes, or we can reduce the data volume and consequently reducing $t_{process}(F_q)$. If we rebalance the data, it is interesting to also rebalance the processing of some branches.

Data reorganization is processed by plugging a dedicated pipeline to the base pipeline, which redirects tuples to the appropriate processing node. After a rebalancing take place, this data reorganization pipelines are removed, and in each node the selection predicates are pushed forward to the data reader to restrict as soon as possible the data volume that it has to gather from storage.

7.4.3 Rebalancing processing data load

When the time required to read the data fragments F_q of a query q, is greater than the time target, $t_{read}(q) > t_{target}(q)$, the current deployment can only provide timely results by reducing the data volume that each node has to process. Therefore, additional processing nodes have to be used in order to reduce $t_{read}(q)$, with data fragments being rebalanced as required, until guaranteeing that query q can be timely processed.

Two distinct approaches can be used: distribute a subset of the fragments required by *WPtree* and replicate *WPTree* and process it in parallel (*splitFQrparallelQr*), or replicate a set of the fragments that are required by a branch and process that branch in parallel (*splitFQparallelQ*).

In *splitFQrparallelQr* (illustrated in Fig. 7.14) the workload processing tree *WPtree* is replicated among processing nodes and a sequence of S'_{Qr}, $S'_{Qr} \subseteq S_{Qr}$, of the fragments required by Q_r are moved (or copied and marked as inactive) to the other node. The execution time of all the queries q, $q \in Q_r$, may benefit with this approach since each node has to process a subset of the initial fragments.

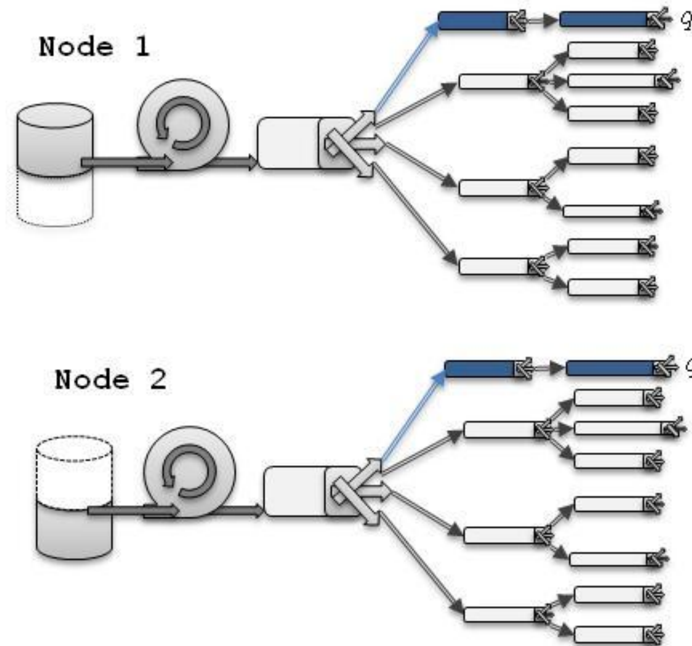


Fig. 7.14 – Distribute Qr fragments and run WPtree in parallel

It tries to evenly distribute the data load among processing nodes. Since the *WPtree* is replicated to every node it is necessary to include an additional merging task for each query that aggregates the partial results computed by each of the nodes. The merging task can be avoided for any query q', $\forall\ q' \in Q_r$, such that $F_{q'} \cap S'_{Qr} = \emptyset$ (none of the relevant data fragment was moved).

In *splitFQparallelQ* (illustrated in Fig. 7.15) a subset $Q_{r'}$ of the queries of *WPtree*, $Q_{r'} \subseteq Q_r$, with $|F_{Qr'}| \simeq |F_{Qr}|$ are chosen to be processed in parallel by processing nodes and then a merging operator is applied to combine its partial results. A subset of the data fragments of $F_{Qr'}$ are then copied into the other node. The execution time of all the queries q', $q' \in Q_r$, which shares common branches with $Q_{r'}$ shares may benefit with this approach since each node has to process a subset of the initial fragments. For those queries, an additional task is required for merging the partial resuls. The remaining queries are just indirectly affected by the skipping of the data fragments. CARROUSEL

dynamically controls when the new node (node 1 in the figure) should process some data fragments on behalf of initial node (node 2).

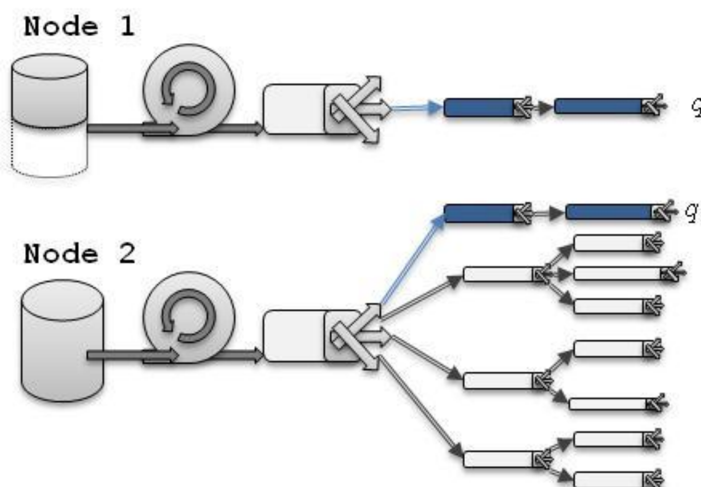


Fig. 7.15 – Replicate a subset of q fragments among nodes and run q in parallel

SplitFQrparallelQr may improve the execution time of all running queries but it adds processing overheads for the merging tasks. *SplitFQparallelQ* may be faster to rearrange, since less data fragments have to be rebalanced among nodes, since it only focuses on the predicates that require more data fragments which prevents the timely execution.

7.4.4 Rebalancing processing load

When the processing time is the constraining factor, a *distributeQr* strategy (illustrated in Fig. 7.16), which selects a subset Q_r' of the processing queries (or just branches), $Q_r' \subseteq Q_r$, such that the processing cost be distributed among processing nodes containing the data fragments $F_{Qr'}$. If the nodes do not have the data fragments required by Q_r', $F_{Qr'}$, they have to be copied. In this case, it tries to evenly divide the branch processing costs, and to select the query subset Q_r' that requires less data fragments to be copied.

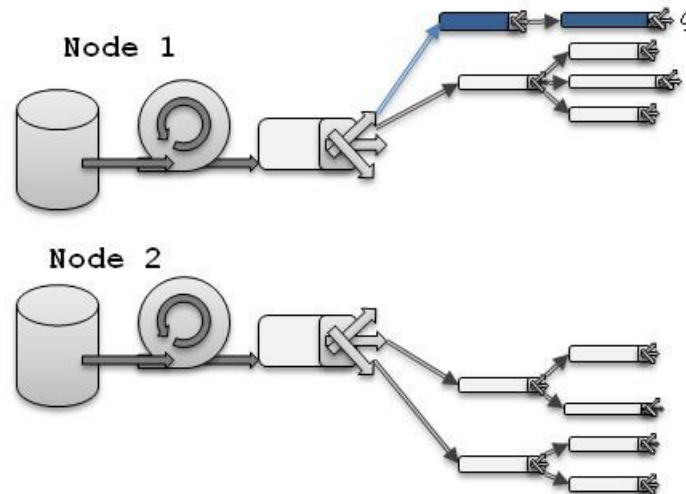


Fig. 7.16 – Split WPtree branches among nodes

Since $t_{process}$ is dependent of the data volume that has to be processed, the data load strategies, discussed above, can also be used to reduce $t_{process}$.

7.5 Fragment level data reorganization

When data fragments have to be rebalanced and copied among processing nodes, CARROUSEL tries to optimize the size of the data fragments and also improve fragment processing (e.g. fragment skipping). It uses the following strategies:

a) ***Tuple reordering within a fragment*** – cluster common tuples together and employ a compression scheme such as RLE encoding [Lemire et al. 2012; Lemire & Kaser 2011] to obtain smaller in-memory bitmaps. To minimize the reordering cost, this process is performed when a similar query is placed at the *WPtree*, or by using other processing node in a parallel deployment.

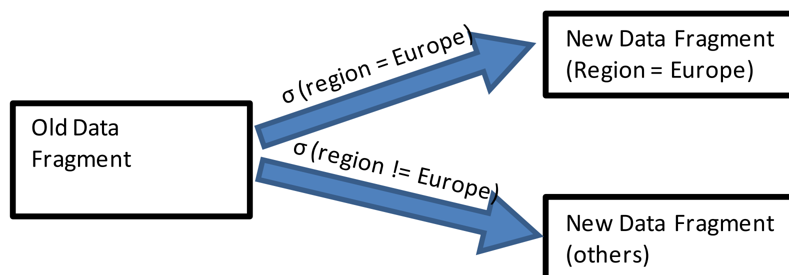
Country	Region	bitmap Region=Europe
UK	Europe	1
USA	America	0
Germany	Europe	1
China	Asia	0
UK	Europe	1
China	Asia	0
China	Asia	0
France	Europe	1

→

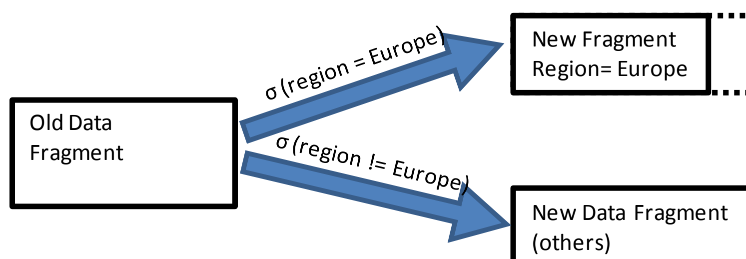
Country	Region	bitmap Region=Europe
USA	America	0
China	Asia	0
China	Asia	0
China	Asia	0
UK	Europe	1
Germany	Europe	1
UK	Europe	1
France	Europe	1

b) ***On-the-fly fragment reorganization*** – a data switch (DS) is placed before more frequent branches to divert more frequently used tuples to new data fragments and the remaining tuples to less used data fragments. Tuples are copied to new fragments based on the current workload pattern to provide

improved performance. The new fragments are added to the *allFragmentsBitmaps*, but they are only considered for execution after being registered in the *activeFragmentsBitmap*. Data fragments are horizontally partitioned on-the-fly into new data fragments with related attribute values. As a result fragment level bitmaps can be built with all “1” or “0” and thus resulting in improved query performance through fragment skipping.



c) ***On-the-fly fragment reorganization with schema compression*** – this strategy is similar to the previous one, but after the reorganization it compresses attributes that, within the data fragment, have low cardinality. Attribute values are replaced with a smaller coded representation. Attributes with a cardinality of 1 (all tuples within the data fragment have the same value), the attribute is removed from the data fragment, and the attribute name and value (e.g. year=2000) is stored at the fragment metadata. Therefore the data fragment will have smaller width, with fewer attributes and smaller attribute values. This besides allowing fragment skipping, it also reduces the fragment’s size and therefore improves performance.



7.6 Chapter Summary

This chapter proposed mechanisms to reorganize the data volume and branch processing of the current deployment, so that it can deliver right-time guarantees.

A method to determine if the current SPIN workload processing tree can provide right-time guarantees is presented. We described a *bitset* processing approach that extends SPIN, which stores the result of predicate evaluations into *bitsets*, and then replaces costly predicate evaluations with fast bit lookup operators. *Bitset* processing reduces the predicate evaluation times and the complexity of the workload processing tree. *Bitsets* also allows the Data Reader to control which data fragments and the order they are read from storage, according to the query time targets.

A parallel SPIN approach, called CARROUSEL, is presented, that manages a set of SPIN processing engines in parallel to speedup query processing and reduce query execution time below the require time targets. Tighter right-time guarantees can be provided by extending the parallel infrastructure and redistributing data among processing nodes, but also by redistributing queries, query processing and data branches among nodes, according to their query load and the data fragments they store.

Chapter 8

Experimental Evaluation

This chapter presents experimental evaluation results of the proposed mechanisms, showing their ability to provide predictable execution times in the presence of large data volumes and large number of simultaneous queries, with almost linear scaleup and speedup. For experimental purposes we used the TPC-H benchmark [TPC-H], a well-known decision support benchmark.

The chapter is divided in the following sections: Section 8.1 describes the experimental setups and the data schemas used in the evaluation; Section 8.2 compares the storage requirements of the ONE data model; Section 8.3 evaluates the predictability and execution times delivered by ONE. In Section 8.4 we evaluate the scalability of ONE when running over a parallel infrastructure (ONE-P) and in Section 8.5 we evaluate TEEPA, its elasticity running over a set of heterogeneous nodes, and its ability to add nodes when necessary until attaining a given time target. In Section 8.6 we evaluate SPIN, the concurrent query processing model, and in Section 8.7 we conclude the experimental evaluation.

8.1 Experimental Setup and benchmarks

This section describes the benchmark, the data schema, the processing infrastructure setup and the database engines used in the experimental evaluation. Unless mentioned otherwise, the results presented in this chapter were obtained as the average of 30 runs for each query, and excluding the two best and worst results.

8.1.1 Benchmark

For experimental evaluation we used the TPC-H benchmark [TPC-H], a Decision Support System benchmark, proposed by Transaction Processing Performance Council. The benchmark defines a set of tables, illustrated in Fig. 8.1 and a set of queries.

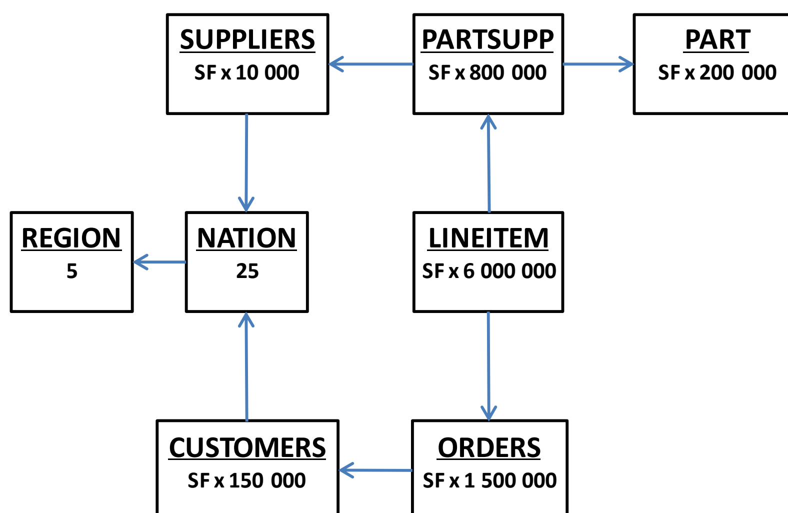


Fig. 8.1 – TPC-H benchmark schema

The benchmark also provides a data generator (DBGEN) for populating the tables, where the data volume is configured with a scale factor (SF) parameter. We used the scale factors 1, 3, 10, 30 and 100. Along the chapter we will use SFx, or SF=x, to denote the scale factor x. Table 8.1 depicts the number of rows for SF1 and the storage size of each table.

SF = 1	Nº Rows	Space
REGION	5	1 KB
NATION	25	5 KB
SUPPLIER	10.000	1,9 MB
CUSTOMER	150.000	31,3 MB
PART	200.000	30,5 MB
PARTSUPP	800.000	164,0 MB
ORDERS	1.500.000	190,3 MB
LINEITEM	6.000.000	726,7 MB
Total		1.144,7 MB

Table 8.1 – Number of rows and estimate size of the TPC-H tables

8.1.2 Data Schemas

With the TPC-H benchmark, we have built the following schemas:

- **TPCH** - the base TPC-H schema as defined in the benchmark, populated with the TPC-H data generator tool (DBGEN) available at [TPC-H].
- **TPCH-P** - the base TPC-H schema deployed on a set of processing nodes where the relation LINEITEM is equi-partitioned with ORDERS and distributed among processing nodes (partitioning alternatives are discussed in Section 2.5). The remaining relations were fully replicated into all the nodes.
- **TPCH-PL** –this schema is similar to the TPC-H schema, except that only the relation LINEITEM is partitioned and distributed among nodes and the remaining relations are fully replicated.
- **ONE -** is composed by a single relation using the ONE data model. It was populated with a modified version of DBGEN that generates the de-normalized data as single file.
- **ONE-P** – is a ONE schema where the relation is partitioned and distributed among a set of parallel nodes. The data volume allocated to each node is adjusted according to the node' characteristics.
- **SPIN –** is a ONE schema processed by the SPIN engine.

For each schema we have generated data for different scale factors, SF = {1, 3, 10, 30, 100}. TPCH(1) and ONE(1), stands for, respectively the TPCH schema and ONE schema, with a scale factor of 1. We also use the notation TPCH1 and ONE1.

8.1.3 Query workload

Since our focus is on providing predictable response times to aggregate queries, from the 22 queries defined in the TPC-H benchmark we used the first 10 queries (1, 2, 3, 4, 5, 6, 7, 8, 9 and 10). We use Qx to denote query x, e.g. Q3 stands for query 3. These queries have different selectivities and complexity and are representative of the query workload. Q1 and Q6 only require data from the LINEITEM, Q2 only processes dimensions, Q3 processes LINEITEM and ORDERS. Appendix A shows TPC-H queries, and Table 8.2 shows which relations are used by each query.

Query / Relation	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10
LINEITEM	X		X	X	X	X	X	X	X	X
ORDERS			X	X	X		X	X	X	X
CUSTOMER			X		X		X	X		X
SUPPLIER		X			X		X	X	X	
PART		X						X	X	
PARTSUPP		X							X	
NATION		X			X		X	X	X	X
REGION		X			X		X	X		

Table 8.2 – Relations needed by each query

With these queries, we defined a set of query loads:

- **Single query load** –each of the above queries is submitted, one at-a-time, by a single client, and thus having exclusive access to all resources of the DB engine.
- **Concurrent query load** – we have a variable number of clients (NClients) that are concurrently submitting queries (chosen in a random manner) to the DB engine.
- **Concurrent query load Q(5)** – this query load is similar to the previous one, except that each client is submitting variants of the query Q5, with random selectivities and selection predicates.

Queries were rewritten according to the used data model. For instance, queries run against the ONE storage model were rewritten to use the de-normalized relation, instead of the star schema model, and the joining conditions were removed.

No specific tuning or tweaking optimizations were made to relations. We did not create any index structures in ONE since we aim to determine the worst case scenario,

where a full table scan is required to process queries, and to determine a maximum invariant time bound for query processing.

8.1.4 Processing Infrastructure

We used the following processing infrastructures:

- **Single Node setup – server (SNS-Server)** – An Intel i5-2500 quad core processor server at 3,70GHz with 8GB of RAM at 1Ghz, with 3 1TB SATA3 Western Digital Blue hard drives plugged to an onboard RAID 0 controller holding the DW data, and 2 other similar hard drives, 1 with the OS and swap and the other with the raw data generated by DBGEN. The server runs a default installation of Ubuntu server 12.04 LTS Linux.
- **Single Node setup – pc (SNS-pc) –** An Intel Dual Core Pentium D processor, at 3.40Ghz, with 2GB Ram, with a 150GB SATA disc drive, and running a default installation of Ubuntu server 10.10 Linux distribution.
- **Parallel Node Setup (PNS)** – The setup for parallel evaluation is composed by 30 processing nodes (15 Intel Core 2 Quad CPU Q6600 @ 2,40GHz with 4GB of RAM, and 15 Intel Core 2 Duo 2,13ghz with 2GB of RAM), running a default Linux server installation. The processing nodes are interconnected with a full-duplex gigabit switch. An additional node was used as the submitter, controller and merger node. Nodes were registered and made available for TEEPA, and added or removed as needed for guaranteeing the time targets.

8.1.5 DBMS engines

For evaluation we used two distinct db engines: one commercial (Oracle 10g release 2) and a open source (PostgreSQL 9.0). We also implemented and used the following prototypes that employ the proposed mechanisms:

- **SPIN (aka SPIN base)** – the base SPIN prototype implemented in Java without bitset processing and carrousel capabilities.
- **SPIN bitset**– the enhanced SPIN setup, which includes run-time bitmap and bitset processing.
- **CARROUSEL** – a parallel deployment composed with a set of nodes, each running a SPIN engine over a set of local data fragments.

8.2 Storage requirements of the ONE data model

In this section we compare the storage requirements of the data models. Table 8.3 shows the storage size occupied by the TPC-H tables, the TPC-H tables with key indexes, and the ONE model for a SF=1.

Data volume (SF1)	Size (MB)	φ_{ss}
TPC-H tables	1.144,7 MB	4,47
TPC-H tables + key Indexes	1.448,4 MB	3,32
ONE	6.270 MB	

Table 8.3 – Storage space required by each schema organization

Since the ONE data model is de-normalized, it requires more storage space than the star schema model. We have defined, in chapter 4, φ_{ss} as the storage space increase ratio in comparison with the TPC-H schemas. ONE represents a 4,47x increase in the storage size in comparison with the base tables, which is reduced to a 3,3x ratio when space required by key indexes is also considered.

For quite some time, this increase in storage size was unacceptable, since disks were expensive, had limited capacity and had slow transfer rates. However, nowadays disks are larger and faster, and can yield sequential transfer rates in the order of hundreds of MB per second, at affordable prices (with prices below 0.05€/GB).

8.2.1 Storage overheads of the star schema model

One of the characteristics of ONE is that it does not need primary and foreign keys. With the purpose of joining relations, the star schema has a set of space overheads related to the set of extra primary and foreign keys and related indexes, which increases the storage requirements. In some DWs the number and space occupied with keys represents a relevant proportion of the overall number of attributes. For instance, the TPC-H schema has 15 attributes (of 61) that are keys, representing a 32% increase in the number of attributes and 7,5% increase in global storage size. Fig. 8.2 depicts the distribution of the storage size occupied by the data (all attributes except keys), the key attributes and the primary key indexes.

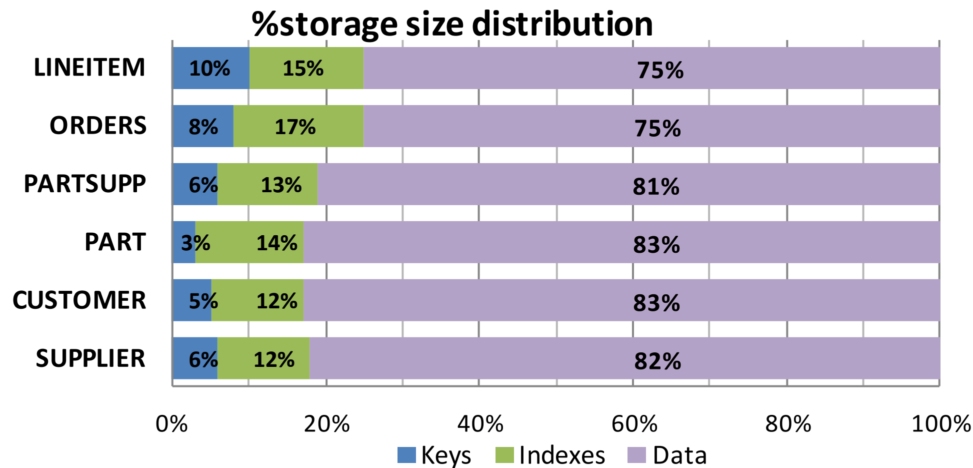


Fig. 8.2 – TPC-H storage size distribution

We observe that keys and related key indexes represent a 25% of the size of tables LINEITEM and ORDERS, and data values represent 75% of the table storage space. When considering other schema, for instance the SSB schema [O'Neil et al. 2009], the key related overhead increases to 70% in both the number of attributes and the storage space.

8.2.2 Storage size in each node of a parallel infrastructure

The deployment of the ONE data model in a parallel infrastructure (ONE-P) tends to be much more efficient than the star schema model. Dimensions typically represent a small percentage of the overall DW storage space. However, in a parallel shared-nothing infrastructure they usually are replicated in each node, and only the fact table is partitioned. Table 8.4 shows the storage space (in each node and the total sum) required by each schema, for a data volume of SF100: ONE-P, TPCH and TPCH-PL. In TPCH-PL only LINEITEM is partitioned and the remaining relations are fully replicated.

In each node				Total of all nodes		
TPCH-P	TPCH-PL	ONE-P	#Nodes	TPCH-P	TPCH-PL	ONE-P
179,4	179,4	627,0	**1**	179,4	179,4	627,0
76,3	91,0	209,0	**3**	228,9	273,0	627,0
40,2	61,1	62,7	**10**	402,2	611,0	627,0
32,5	54,7	31,4	**20**	649,7	1.094,0	627,0
29,9	52,6	20,9	**30**	897,2	1.578,0	627,0
27,8	50,9	12,5	**50**	1.392,3	2.545,0	627,0

Table 8.4 – Storage space required by each schema organization for SF100 (in GB)

We observe that the overall size of ONE-P remained constant, regardless of the number of nodes, whilst the TPCH-P requires more storage space, for instance in a 30 nodes setup, the overall size of TPCH-P increased by a factor of 5. Query execution time with ONE is volume-dependent, thus the total execution time with a ONE-P(30) setup is linearly reduced to about 1/10th of the measured execution time with a 3-node setup, excluding the inter-node data transfers and merging costs. Considering the number of tuples in each node, in a 30 node setup the dimension PART (200k rows) has the same number of tuples as LINEITEM (6M rows/ 30= 200k rows), CUSTOMER (150k rows) has 3x more tuples than ORDERS (1,5M rows / 30 = 50k rows), and PARTSUPP (800k rows) 4x more tuples than LINEITEM (6M rows / 30=200k rows).

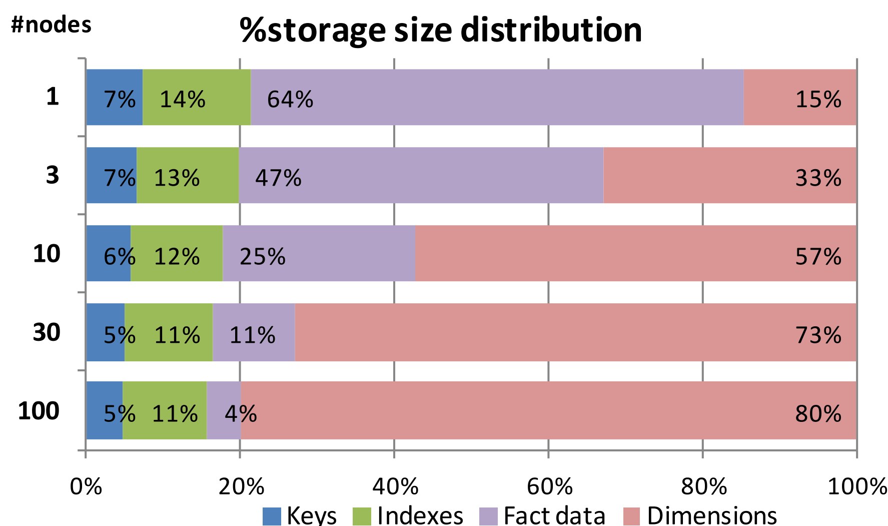


Fig. 8.3 – Storage size distribution with different numbers of processing nodes (TPCH-P))

The decrease of the fact data allocated to each node, as we increase the number of nodes, changes the data distribution in each node, as illustrated in Fig. 8.3. The percentage occupied by dimensions increases with the number of nodes, impacting the way queries are processed. Dimensions (which typically are smaller than fact tables), which usually are used as the inner relation in hash join operations, but in large parallel infrastructures they tend to be outer relations. The query execution plans tend to be more dependent of the cost to process dimensions, as we increase the number of nodes, and thus limiting its scalability.

We calculate storage scalability as the ratio of *size*(*1-node*) / ∑ *size*(*n-nodes*). Fig. 8.4 depicts the results of the storage scalability for different schemas.

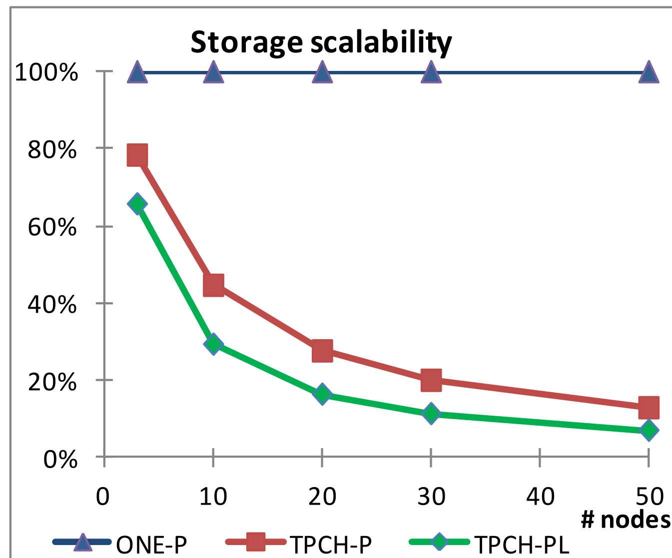


Fig. 8.4 – Storage scalability of the schemas

The results show that ONE-P provides linear storage scalability. The scalability of the other schemas is constrained by the size of replicated dimensions and the key related overheads, as illustrated in Fig. 8.3. For instance, with a 10 nodes setup, TPCH-PL yields around 30%, this means that each node requires almost 1/3 of the single node storage requirements.

8.3 Evaluation of Execution Time of ONE

In this section we evaluate the effectiveness of ONE in providing predictable execution times. For the experimental setup, we used a Single Node Setup (SNS-server) infrastructure and the TPCH and ONE data schemas with scale factors of 1,3,10 and 30.

8.3.1 Predictable performance with scalable data volumes

In this section, we evaluate the execution time predictability of both models, using a single node setup, when the data volume increases from SF1 to SF10 (a 10x increase in data volume). Fig. 8.5 depicts the results for TPCH. The primary vertical y-axis (on the left side) represents the execution time scale for SF1, while the secondary vertical y-axis (on the right) represents the execution time scale for SF10. The scales are set so that the scale of SF10 is 10× the scale of SF1.

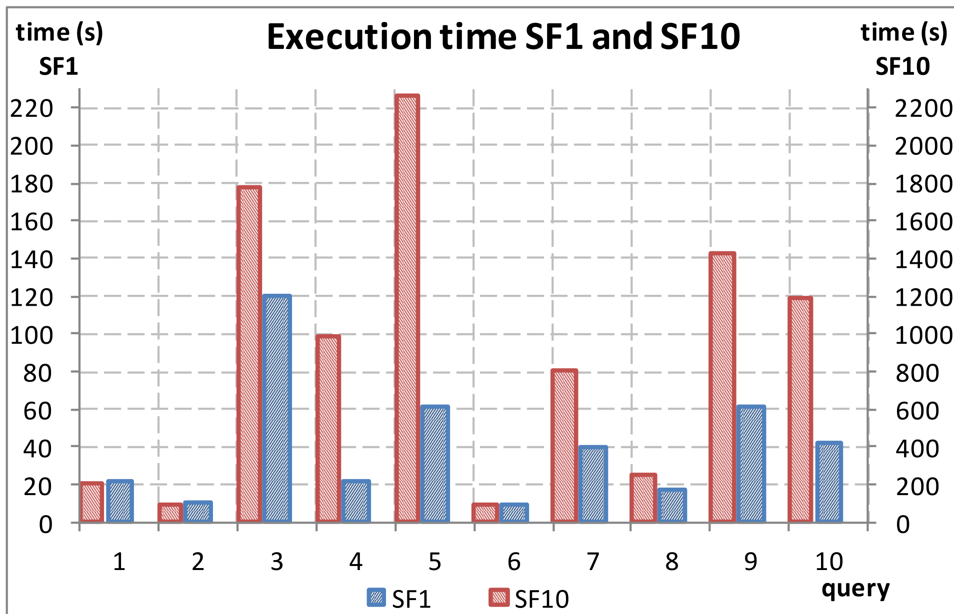


Fig. 8.5 – TPCH execution times for large data volumes

From the figure, we observe that the query execution times of SF10 are not proportional to those of SF1, given the scale factor. While the results for 3 of the queries (Q1, Q2, Q6) have similar behavior, the remaining queries behave differently with different data volumes. The execution time is not predictable, since it varies from query to query (each query returns a different execution time), and it also varies in an unpredictable manner with data volumes (SF10 - in red - does not follow the pattern of SF1 – in blue). The degradation of execution times is much bigger than the increase in data volumes.

The execution time results for ONE are depicted in Fig. 8.6. As in the previous figure, the primary vertical y-axis represents the time execution scale for the SF1, while the secondary vertical y-axis represents the time execution scale for SF10. The scales are set so that the SF10 scale is 10× the SF1 scale.

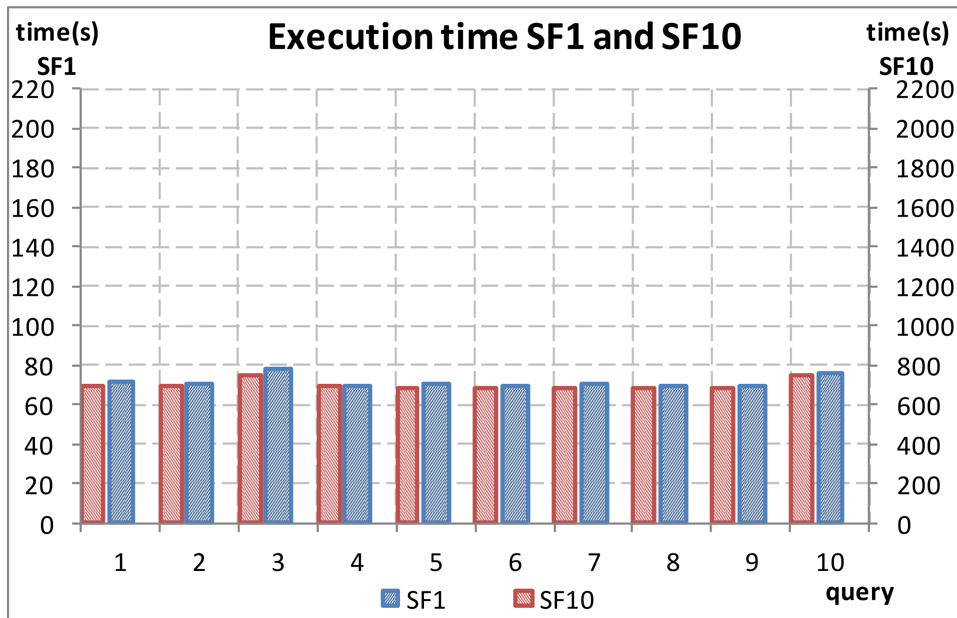


Fig. 8.6 – ONE predictable execution time for larger data volumes

From the figure, we observe that with ONE the execution time for a given data volume is almost the same for all the queries. The execution time of query 3 is slightly higher because it returns a large number of aggregation groups and the presented results were gathered at client (i.e. it also accounts for the network time of sending the results).

Another observation that stands out is that, for larger data volumes, ONE provides a predictable execution time, i.e. the SF10 bar (in red) is almost similar to the SF1 bar (in blue). They are not equal, because for each query there is a set of processing costs that are independent of the data volume. The execution time of SF10 is less than the execution time of 10 times SF1 (SF10 ≈ 10×SF1).

Using the execution times of SF1 to predict the execution time for larger data volumes, ONE yields more accurate execution time estimation than TPCH. Fig. 8.7 depicts the estimation error in predicting the query execution time for larger data volumes for both models. The estimation error was calculated as

$$(t_{exec}\,(\text{SF10}) - 10\times\, t_{exec}\,(\text{SF1})\,)\,/\,t_{exec}\,(\text{SF10})$$

	1	2	3	4	5	6	7	8	9	10
ONE	-3,0%	-0,4%	-4,5%	-0,1%	-2,3%	-2,1%	-2,2%	-2,1%	-2,1%	-1,4%
TPCH	-4,2%	-10,9%	32,0%	78,2%	72,8%	0,9%	49,9%	31,3%	57,2%	64,3%

Fig. 8.7 – Error estimation in predicting the query execution time for higher data volumes

The results show that the estimation error of ONE is always below 0, i.e. the real execution time is below the estimation based on the execution time of the smaller data set. On the other hand, the real execution time for larger data volumes is typically underestimated, in some cases it requires more than 70% of the estimated time.

8.3.2 Execution Time (single node with Oracle)

We evaluated both setups using scale factors {1, 3, 10, 30, 100} for queries Q1,…,Q10. In each setup, we run each query 30 times to obtain the query execution time results, and excluded the two best and worst execution times for each query. Fig. 8.8 shows the average of the execution times with different scale factors.

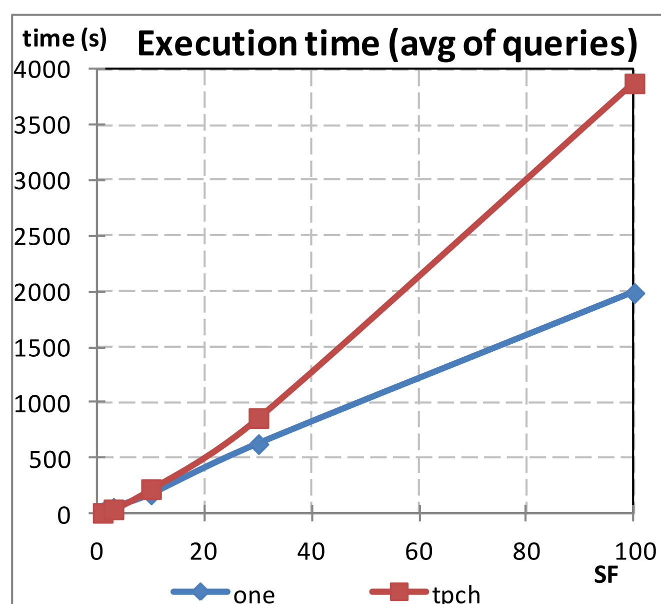


Fig. 8.8 – Average execution time for queries 1 .. 10

With small scale factors, TPCH may, on average, deliver better execution times, since joins are mostly performed in memory. But as the data volume increases, some queries require more than the available memory to process some operations such as joins, and have to use temporary storage for executing them.

From the figure 8.8, it is clear that the execution time of ONE increases linearly with the data volume, depicting an almost perfect line. For larger data volumes, ONE delivers better performance results than the star schema approach, and with predictable execution times. This is possible because of its simpler query execution plans, without joins, and thus it is fairly independent of the amount of available memory. Fig. 8.9 depicts the execution time for each query for different scale factors.

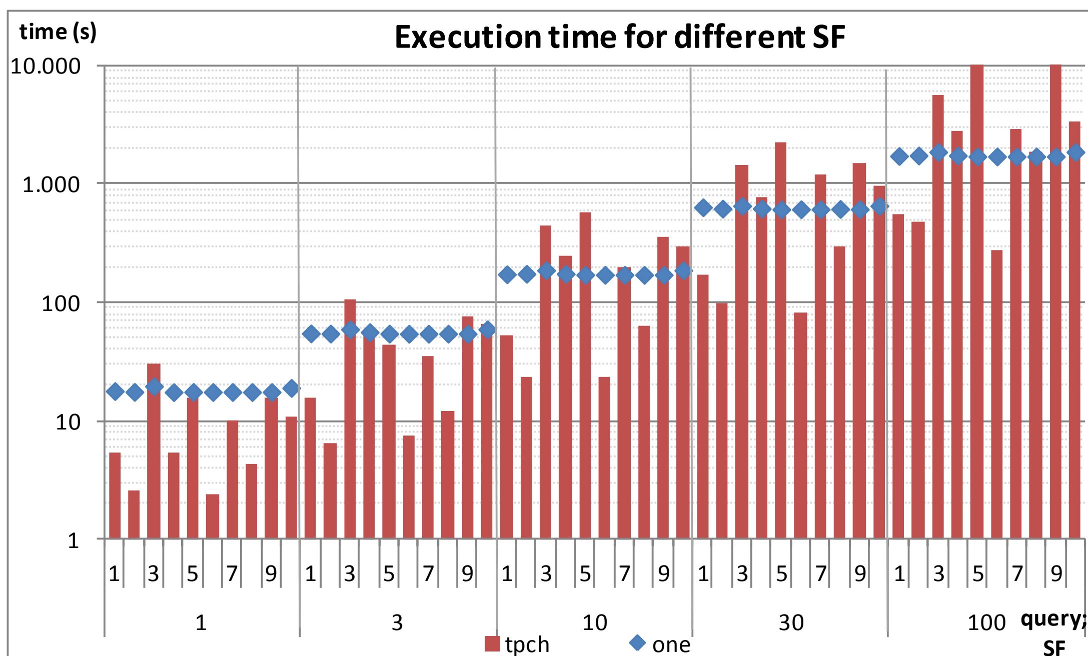


Fig. 8.9 – Average execution time of ONE for varying SF for queries 1-10 (Oracle)

From the figure, one thing that stands out is that ONE, for the same scale factor, presents query execution times with small variability, while the TPCH schema shows a much larger variability. Another interesting aspect is that for small data volumes, as expected, most queries run against TPCH exhibit faster execution times than with ONE. This is because a large proportion of dimension data can fully reside in memory and there is enough memory to perform in-memory joins. However, for larger data volumes, the available memory is insufficient to hold dimension data and to allow in-memory joins. When this happens we witness a drop in performance because of the expensive IO operations (including some random read and write operations) required to perform joins.

ONE may not always deliver the fastest results, since some queries show worse execution times, when compared with TPCH, however ONE delivers better execution times for large data sets and its execution times are almost constant and predictable. We will show later that adding parallelism to ONE will make it even much faster than the traditional approaches, while also maintaining predictability of query execution times.

8.3.3 Execution time variability

Besides the average execution time, we also evaluated the variability of the query execution times. The results are depicted in Fig. 8.10 .

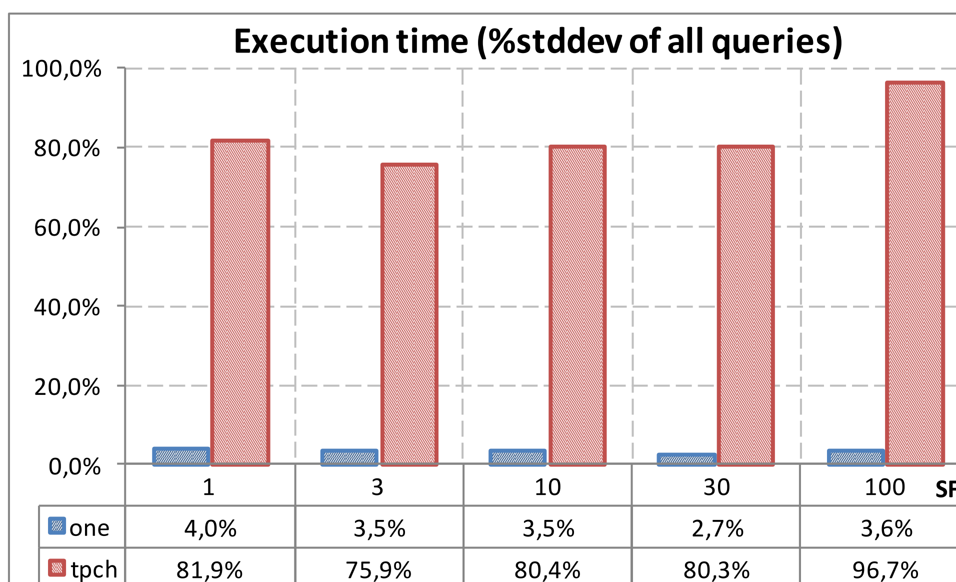


Fig. 8.10 – Execution time variability for varying SF

ONE provides good standard deviation, as illustrated in Fig. 8.10, resulting in low execution time variability (less or equal to 4% of the average execution time) in all analyzed scale factors (1, 3, 10, 30 and 100), demonstrating ONE' capability to execute queries with a predictable execution time. TPCH, on the other hand, provides a minimum of 75%, with a scale factor of 3 and reaching up to 96% of the average execution time with a scale factor of 100. Furthermore, the standard deviation of ONE is impressive.

While TPCH performs better, at small scale factors, since a large amount of the inner relations resides in memory, requiring less IO operations, the query execution time is highly unpredictable. ONE, starting from a scale factor of 10 (SF=10), presents on average, faster query execution times than TPCH (see Fig. 8.8 and Fig. 8.9).

8.4 Evaluation of ONE-P

In this section we evaluate the ONE data model in a parallel deployment (ONE-P), using 3, 10, 20 and 30 nodes. An additional node was used as the submitter, controller and merger node. The evaluation results were obtained with a data volume of SF=100. The data volume allocated to each node and the overall storage requirements, considering all processing nodes, are depicted respectively in Fig. 8.11 a) and b).

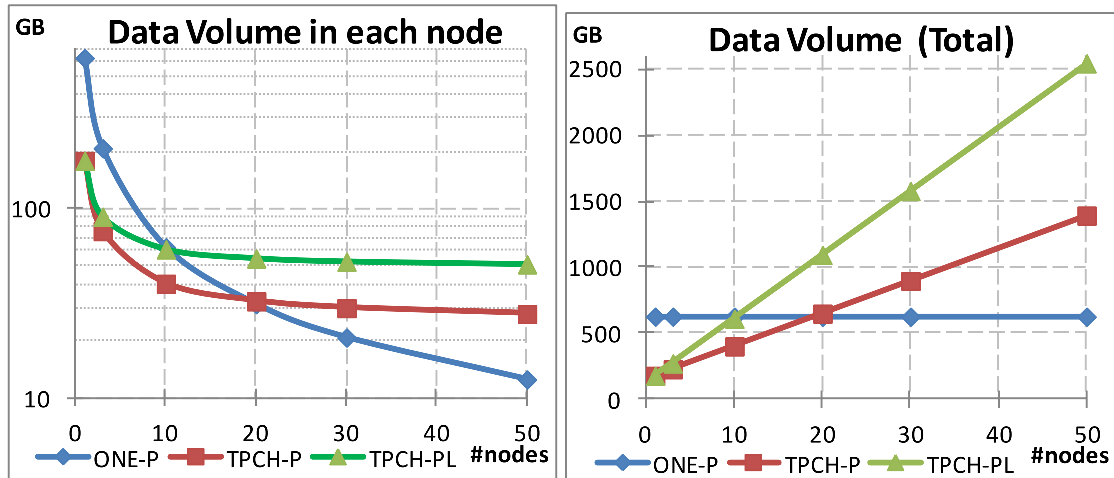


Fig. 8.11 – Data volume a) in each node b) total (sum each node)

With TPCH-P, as shown in Table 8.4, the data volume in each node does not decrease proportionally with the number of nodes, as ONE-P does. In a setup with 30 nodes, each node stores about 16% of the base data volume, and the overall storage space (sum of all the nodes) increased by a factor of 5, while the total size of ONE-P remains constant, regardless of the number of nodes.

8.4.1 Execution time (in each node)

In a parallel shared nothing infrastructure composed with n nodes, the query execution time is obtained as the sum of several partial times (recall definition 5.1), some of them are not dependent of the used data model. For instance, for a query *q*, the time related to the exchange and merging of partial results sent by each node is roughly the same, regardless of the time that each node required to process ONE-P or TPCH-P. The query is the same (except the minimal changes to select the appropriated relations) and return the same amount of partial results. For that reason, we will focus on evaluating the local execution time in a node (t_n), which may vary when using different data models, in a parallel architecture composed with 3, 10, 20 and 30 nodes for a scale factor SF=100. We evaluated the average time that a node needs to compute locally the

query partial results. Fig. 8.12 depicts the average execution time for a varying number of nodes.

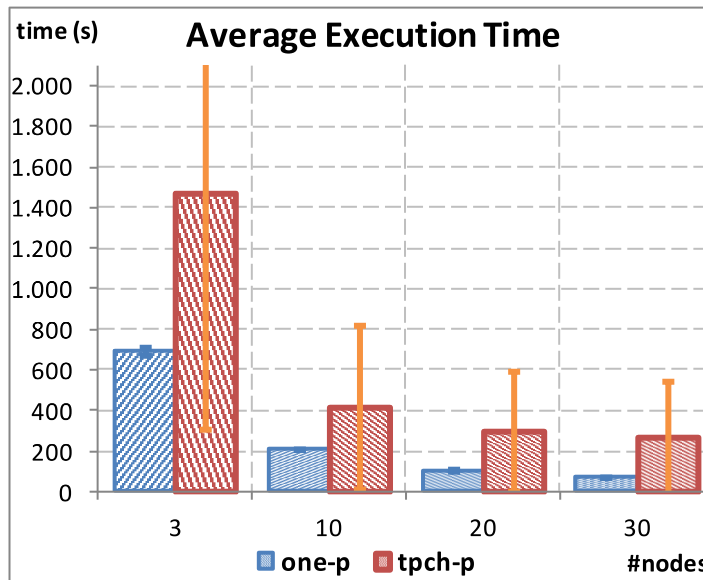


Fig. 8.12 – Average time to process partial results (t_n) in each node

We observe that the local execution time of ONE-P decreases almost linearly as we increase the number of nodes, and consequently the data volume that each node has to process is reduced proportionally. The execution times of TPCH-P is also improved but at a lower rate. Fig. 8.13 shows, for a varying number of nodes, the average execution time required by the nodes to compute the partial results of each query.

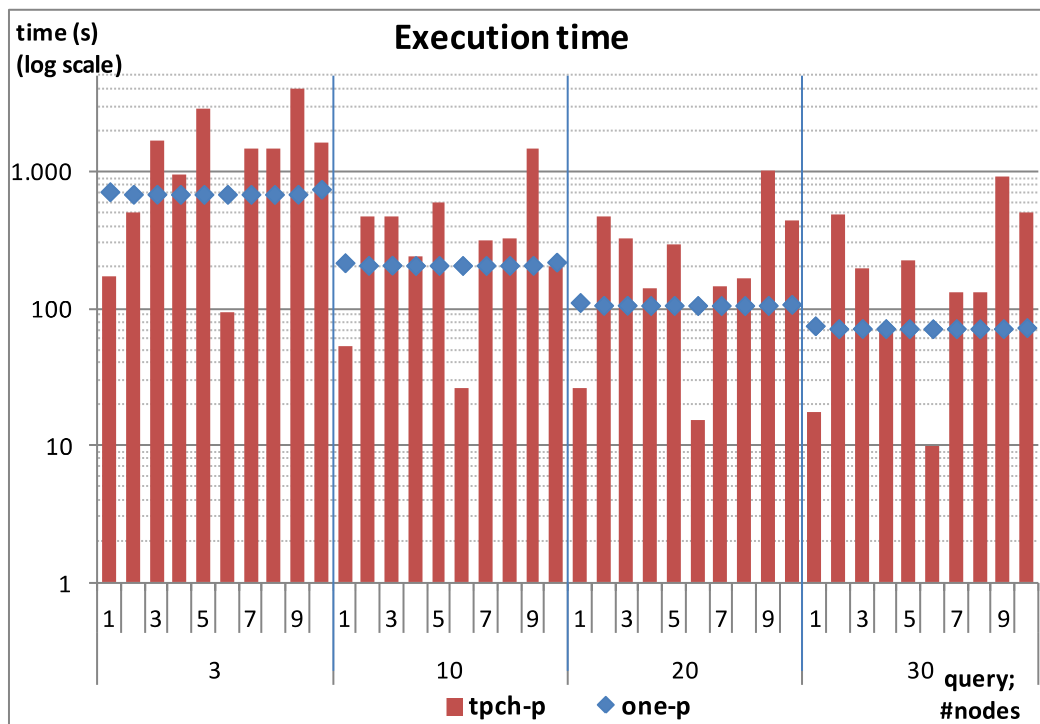


Fig. 8.13 – Execution time to compute the partial results of Q1..10

One thing that stands out is the low variability of the execution time delivered by ONE-P, while with TPCH-P the execution time varies greatly from query to query. TPCH-P only yields better results than ONE-P for queries Q1 and Q6, that only process table LINEITEM, which is partitioned among nodes and the partition size is smaller than the ONE partition. For the remaining queries, ONE-P presents faster execution times for all considered parallel setups. And its execution time is reduced proportionally to the number of processing nodes, since data can be fully partitioned among nodes. ONE-P delivers good performance results and it is unaffected by query selectivity.

8.4.2 Speedup with larger processing infrastructures

Because the data with TPCH-P is not fully partitioned, dimensions are replicated or partially equi-partitioned, an increase in the number of processing nodes does not yield a proportional impact in query execution time. As a consequence, the speedup of TPCH-P is sub-linear. Considering the query execution times of queries Q1..Q10, we depict in Fig. 8.14 the speedup of TPCH-P and ONE-P.

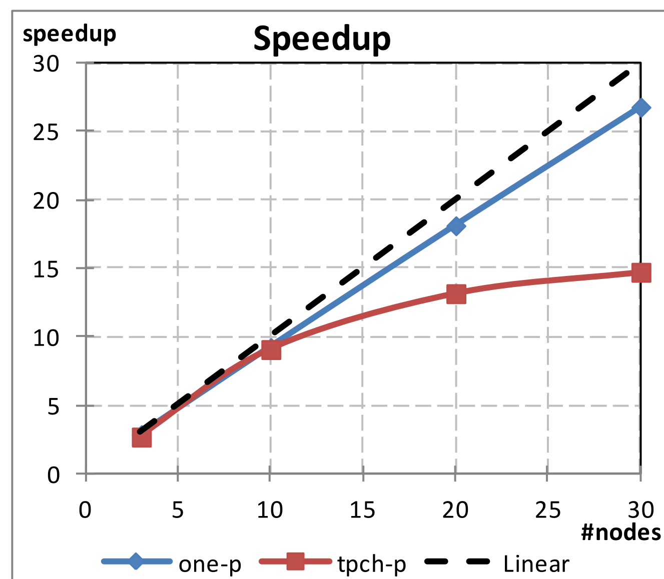


Fig. 8.14 – ONE-P and TPCH-P speedup

The results show that ONE-P delivers an almost optimal speedup, because of the processing costs related to the exchange and merging of partial results, while TPCH-P delivers a sub-linear speedup. In a setup with 30 nodes we observe that the speedup delivered by ONE-P is almost two times greater than with TPCH-P.

8.4.3 Impact of query selectivity in performance

ONE-P, in contrast with TPCH-P, is immune to the joined query selectivity. For instance considering query Q5, which lists the revenue volume done through local suppliers in a given year, if we increase the time interval from 1 year to 2, 3 or 4 years we observe an increase in execution time with TPCH-P, while with ONE-P the execution time remains unchanged. In Fig. 8.15, which depicts the impact of changing the selectivity in query execution time, we observe that the execution time of query Q5 with a 4 year time interval (represents a selectivity of 11%) increased up to 25% in a setup with 3 nodes.

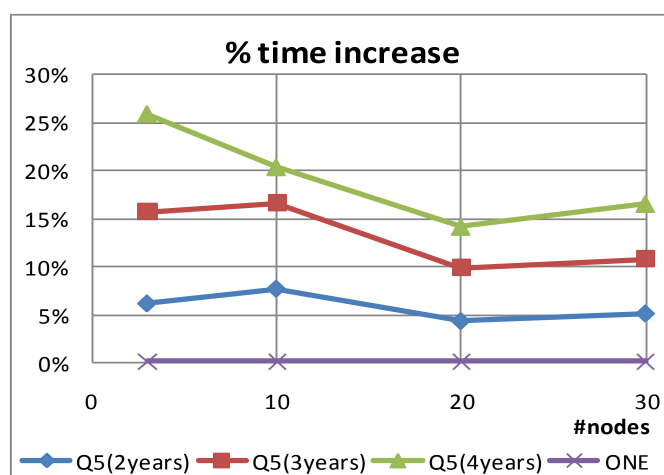


Fig. 8.15 – %time increase for Q5 with different selectivity

ONE-P is not affected by the changes in selectivity since the number of aggregations groups remained the same, only the number of tuples that fall in each group have changed.

8.4.4 Inter-node variance of query execution time

Beside the average time that nodes needed to compute the partial results, over the data stored locally, we also measured the variation of the execution times amongst processing nodes. The results for query Q5 are depicted in Fig. 8.16.

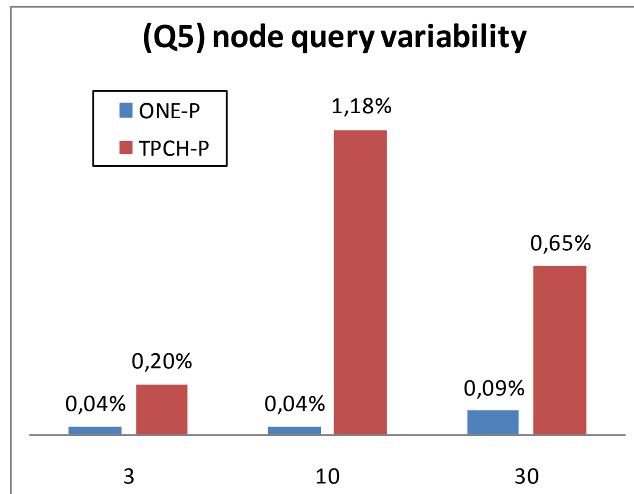


Fig. 8.16 – Q5 node query variability

We observe that ONE delivers low query variability, for all the considered parallel setups (3, 10 and 30 nodes) it remained below 0,1%, with TPCH always delivering higher results, reaching up to 1,18% with 10 nodes.

8.4.5 Cost of exchanging partial results

Query processing in a shared nothing parallel architecture as shown in equation 1 (in Section 5.1) is influenced by the local partial query execution time (t_{nj}), and also by the cost of merging (t_m) and exchanging the partial results (t_{tpr}), whose costs varies with the size of the partial results and the number of processing nodes. We considerer negligible the time required to rewrite the query (t_{rw}) and to transfer the partial queries (t_{tpq}). The cost of exchanging the partial results may be large and it depends on the query and the number of nodes. We used TPCH-P (ONE-P returns the same amount of partial data) in parallel setup composed with 30 nodes to measure the time needed to exchange the partial results of each of the queries (Q1,..,Q10). The results are depicted in Fig. 8.17.

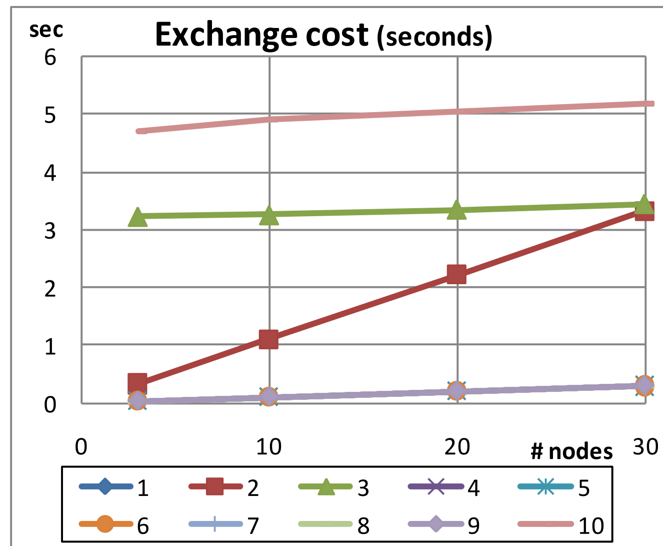


Fig. 8.17 – Total exchange of the partial results (t_{tpr}) for queries Q1..Q10

With the exception of queries Q2, Q3 and Q10, the data exchange time has reduced impact (below 1 sec) in the overall query execution time. The exchange time of Q2 is almost linear, since the size of the partial result computed by each node is almost the same. Therefore a larger processing infrastructure produces more partial results that have to be sent to the merger node. On the other hand, queries Q3 and Q10 are almost invariant to the number of nodes, since the aggregations groups are almost fully partitioned among nodes. In general, the exchange overhead can represent up to 8% (Q2, Q10) and 5% (Q3) of the local query execution time.

8.5 TEEPA Right-Time Evaluation

In this section we evaluate TEEPA and its ability to provide right-time execution times, by changing the number of nodes of the parallel processing infrastructure and rebalancing the data volume among nodes, when the time targets cannot be guarantee. To serve as baseline, we used TEEPA to manage a data volume of SF=100 over a TPCH-P deployment composed with 3 nodes. Fig. 8.18 depicts the average partial execution time for each query.

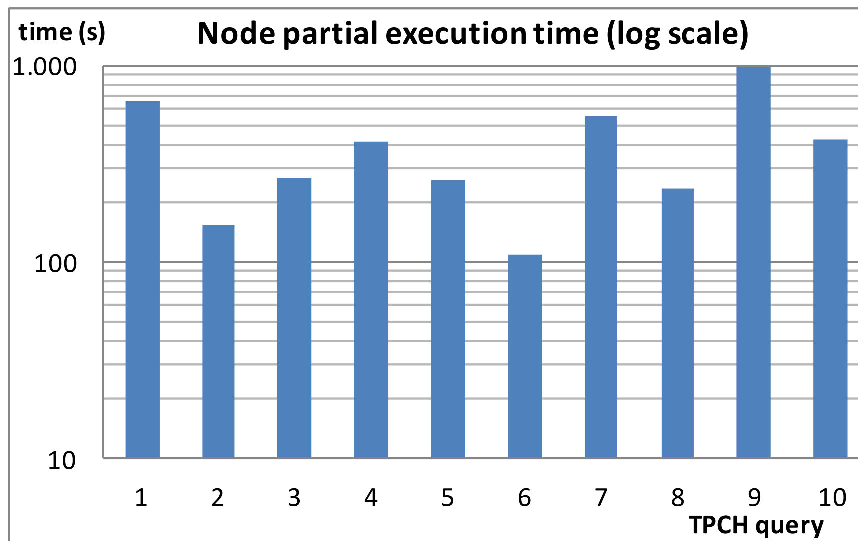


Fig. 8.18 – Partial execution time of TPCH per query in each node

Since we used TPCH-P, queries exhibit different partial execution times. When TEEPA is running, it evaluates the current star schema deployment and the predictable performance of ONE to determine which data model provides improved performance. Fig. 8.19 depicts both the current star schema performance, and a performance estimation of the ONE-P deployment. Considering the maximum execution time (max time bound) required by each data model, TEEPA selects the data model that has least value (depicted in the figure as a green line).

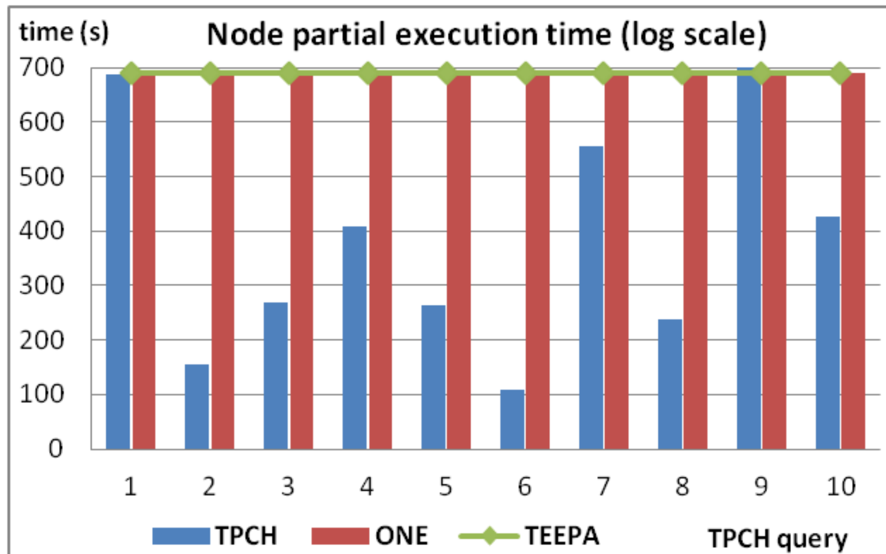


Fig. 8.19 – TEEPA partial execution times (3-node setup)

Since no time targets were defined, and TEEPA determined that the current storage model offers improved performance, the used data model was left unchanged. To evaluate TEEPA elasticity and its ability to provide timely executions, we set the session time execution target to 250s.

```
ALTER SESSION SET TIME_TARGET WITHIN 250 seconds;
```

With this target, more than 2/3 of the queries fail the time target in the current parallel deployment composed with 3 nodes. TEEPA determined that the parallel infrastructure needs at least 10 nodes in order to be able to timely execute the queries, before the time execution target.

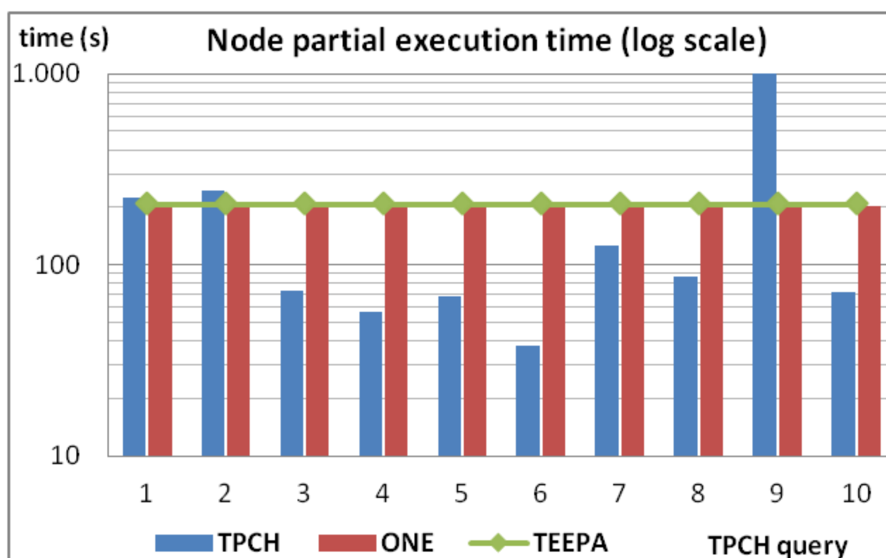


Fig. 8.20 – TEEPA partial execution times (10-node setup)

TEEPA extended the processing infrastructure from 3 to 10 nodes, with the addition of 7 new nodes, and rebalanced and distributed the data volume among them. The query execution time for this deployment is depicted in Fig. 8.20. TEEPA realized that the execution time of 3 of the queries (Q1, Q2 and Q9) run against the star schema model are not within the time target. And therefore additional nodes have to be added or the data model has to be changed. It realized that the execution time estimated for the ONE data model, is below the specified time target (green line). Therefore it triggered a de-normalization process to switch to the ONE data model. TEEPA took around 10 minutes to include 7 additional nodes, to balance the data among the 10 nodes and to reorganize the data into a de-normalized schema, before start delivering timely results within the defined target.

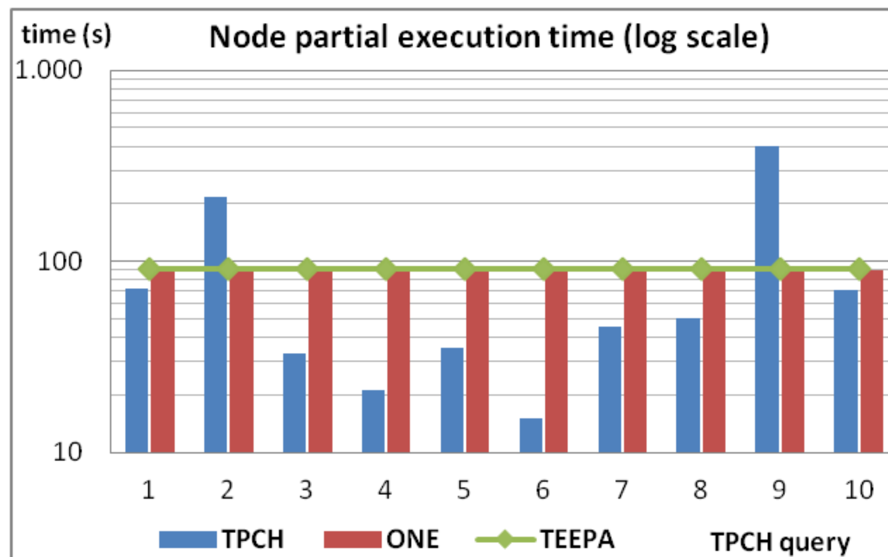


Fig. 8.21 – TEEPA partial execution times (30-node setup)

Afterwards, we defined a tighter time target of 100s. TEEPA then determined that at least 30 nodes were needed to provide such timely execution guarantees. It took about 4 minutes to make all the necessary readjustments, including the addition of 20 nodes and the redistribution the data from the initial 10 nodes to the new ones, before starting to deliver timely query results. In this case, it decided to maintain the ONE data model. Fig. 8.21 shows that the partial query execution times with the new processing infrastructure are below the 100 seconds time target. The variance of the execution times remained below 0.05%.

8.6 SPIN Evaluation

This section shows experimental evaluation results obtained with the SPIN prototype (release 1.6.3). These results were obtained with the default data reader that assumes the ONE data model and handles tuples physically stored in a row-wise format, without optimization features (e.g. compression, materialized views, automatic in-memory bit-selections).

To evaluate the impact of concurrent query loads in performance and scalability, we evaluated the TPCH setup with a query load composed by several variants of the query Q5 with different selectivity and aggregation groups. To evaluate the influence of the workload query pattern in the average execution time, in Section 8.6.4, we use a more complex workload composed with variants of the queries Q1, Q3 and Q5 with distinct query predicates. The query load was generated by a varying number of simultaneous clients that submit a total of 1000 queries, chosen randomly among the variants. The depicted results were obtained as average of 30 runs.

8.6.1 Influence of number of queries in query performance

The query execution time of common RDBMS that follow a query-at-time execution model is highly influenced by the number of queries that are concurrently being executed. In this setup, we evaluate how the number of concurrent queries influences the average execution of SPIN and TPCH. Fig. 8.22 depicts the average execution time for a scale factor of SF=10.

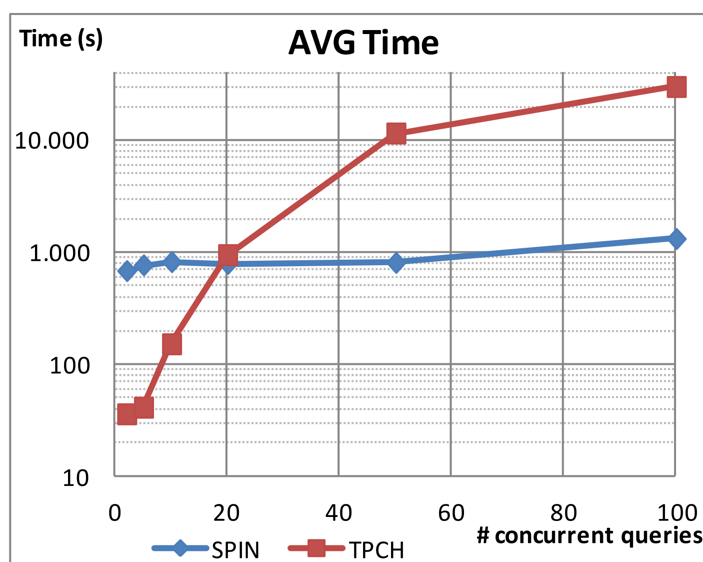


Fig. 8.22 – Average execution time for varying query loads (lower is better)

We observe that at low concurrent query loads (less than 20 queries being executed simultaneously), the TPCH setup yields better average execution times. However, as the number of concurrent queries increases, TPCH exhibits significantly higher average execution times, because more queries are competing for resources. On the other hand, the average execution time with the SPIN setup remains almost constant. There is a slight increase at higher concurrent query loads due to the pipeline management overheads and the cost of processing the query-specific pipelines that cannot be combined with other query pipelines.

The impact of submitting additional queries in the average execution time of the running queries is depicted in Fig. 8.23. The results show that at higher query loads, SPIN introduces low overheads per query (below 1%) in the average execution time. The overhead is higher at low query loads (less than 10 concurrent queries) because the running queries exhibit fewer opportunities for sharing data and processing.

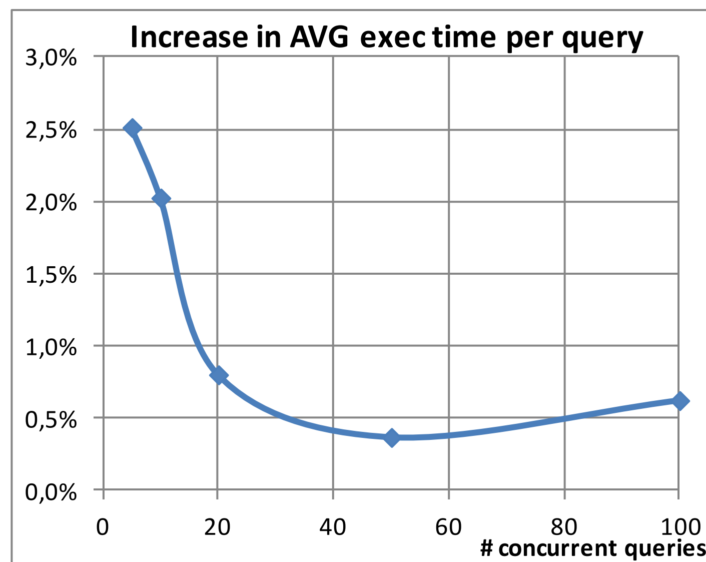


Fig. 8.23 – Overhead per query in the average execution time (lower is better)

The results also reveal that, when a query is submitted using SPIN, there is a high degree of confidence regarding how long it will take to deliver the result, regardless of the currently running query load.

8.6.2 Influence of number of queries in Throughput

Since the average execution time is not significantly influenced by the concurrent query load, SPIN yields almost linear throughput. The probability that two or more queries may share overlapping selection predicates and processing operators, increases as more queries are running at the same time and consequently yielding improved throughput. Fig. 8.24 depicts the throughput in queries per hour (Qph) achieved by TPCH and SPIN.

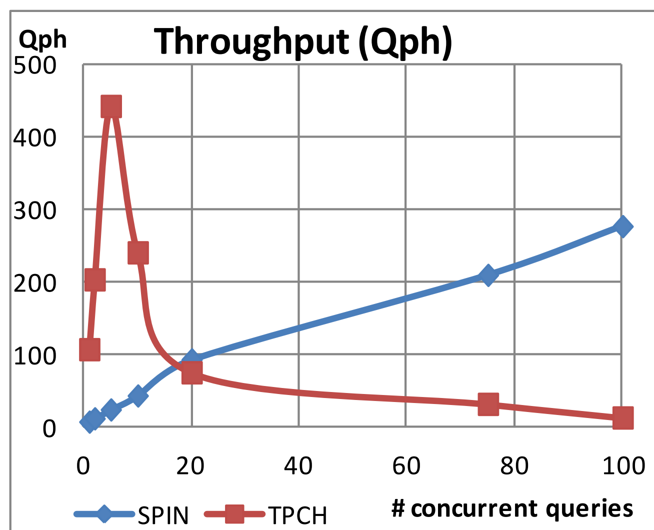


Fig. 8.24 – Throughput for varying query loads (higher is better)

In the figure we observe that at low concurrent query loads, TPCH yields higher throughput than SPIN. However as we increase the number of queries that executed simultaneously, this behavior changes drastically. SPIN does not deliver a linear throughput due to the pipeline management overheads and also the cost of processing the query-specific pipelines that cannot be combined with other query pipelines

8.6.3 Influence of the data volume in throughput

Throughput is influenced by the query load, but also by the data volume. Fig. 8.25 depicts the throughput for two distinct data volumes: SF1 (a) and SF10 (b).

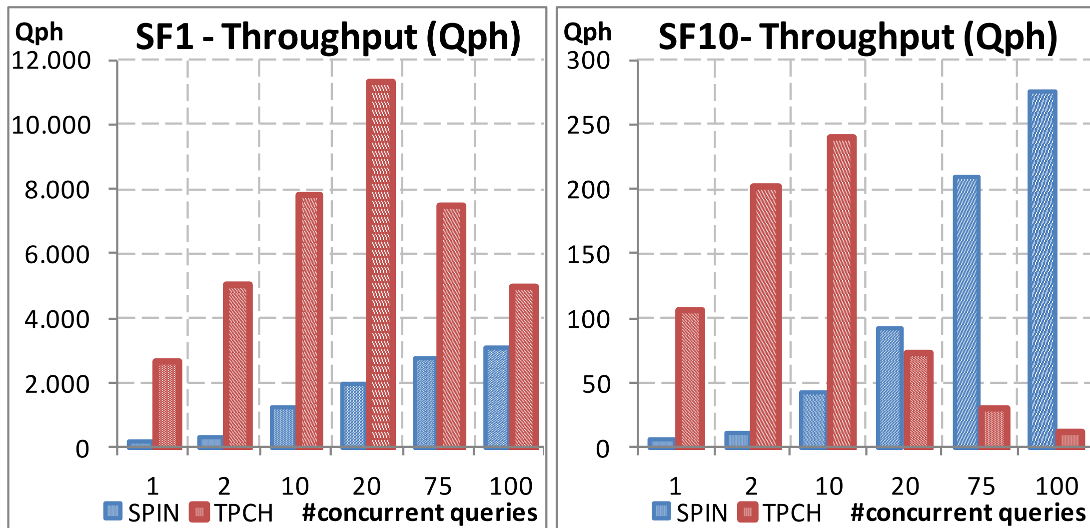


Fig. 8.25 – Throughput for varying query loads with a) SF =1 and b) SF =10 (higher is better)

In the figure, we observe that throughput of SPIN increases almost linearly with the increase of the number of queries that are ran simultaneously, as more data and processing is shared among them. With SF1, TPCH yields significantly higher throughput since all data and processing is done almost exclusively in memory. However as the number of concurrent queries increase we observe a significant drop in throughput as the running queries exhaust the available memory. SPIN, does not have these memory issues and can be massively deployed among low-end commodity servers.

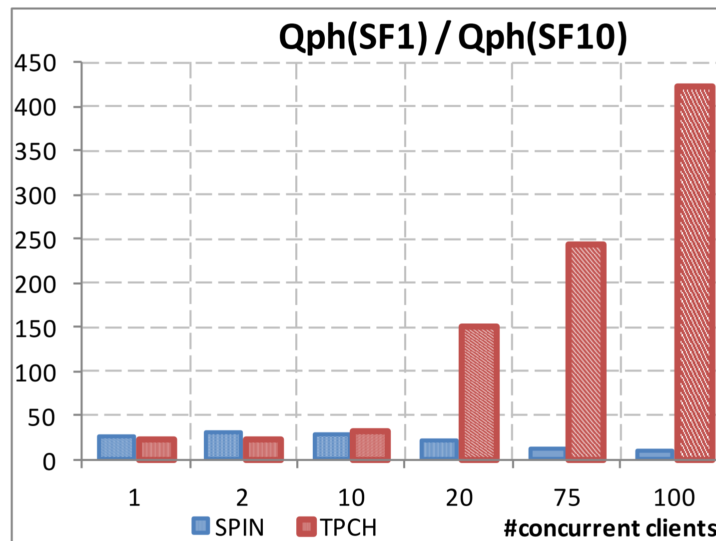


Fig. 8.26 – Impact in throughput of a 10x increase in data volume (lower is better)

This effect can be observed in Fig. 8.26, which compares the throughput ratio (Qph(SF1) / Qph(SF10)). As the data volume increases by a factor of 10, from SF=1 to SF=10, we observe that at low query loads (less than 10 queries running concurrently), the throughput drops by a factor of around 30, mainly because with SF=1, TPCH is

almost entirely in memory, while with SF=10 there is more IO operations. As the query load increases, the throughput of TPCH drops, since more IO operations are needed to process the queries. With 100 concurrent queries, an increase in data volume by factor of 10 results in a drop in throughput by a factor greater than 400. On the other hand, the throughput of SPIN drops almost proportionally to the data volume increase factor.

8.6.4 Influence of the workload query pattern in query performance

The above experiments were carried out using variants of query Q5 with different selectivity and aggregation groups. To evaluate the influence of the query load pattern in the average execution time of SPIN, we now evaluate more complex query loads composed with variants of the queries Q1, Q3 and Q5 with distinct query predicates.

Fig. 8.27 depicts SPIN results for SF=10 with three distinct query workloads: Q1 and Q5 are query workloads exclusively composed by variants of query Q1 and Q5, respectively; Q135 is a workload composed by a set of variants of the queries Q1, Q3 and Q5, that where randomly chosen for execution.

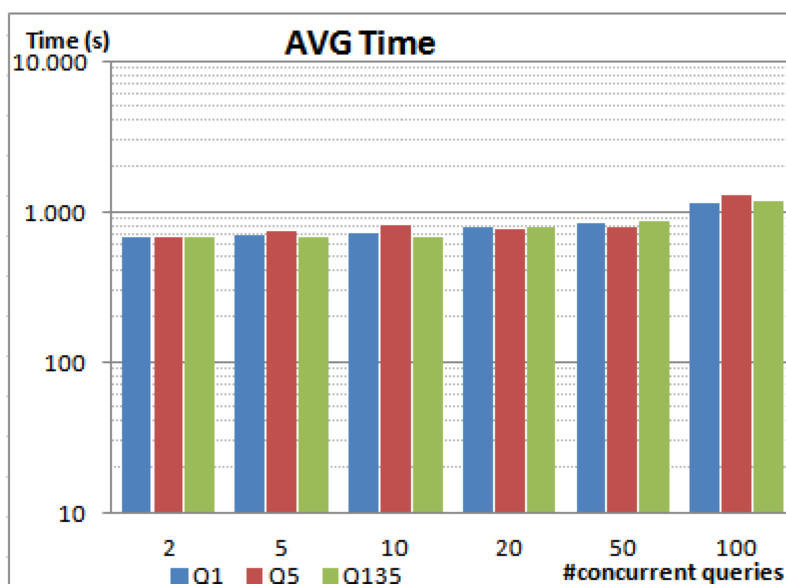


Fig. 8.27 – Influence of the query workload pattern in average execution time

The results show that the query workload has minimum impact in the average execution time, for all the considered number of concurrent queries. Although some queries (e.g. Q1) have more complex calculus, there is no significant change in the average execution times because CPU is not the bottleneck, but the IO performance.

Therefore, and because it uses the scalable and predictable ONE data model, SPIN performance can be significantly boosted if a parallel infrastructure is used.

8.6.5 Evaluation of SPIN with bitset processing

We selected a query load, based on variants of query Q5, which SPIN already delivers good performance results, to assess if *bitset* processing can improve SPIN even further. The workload processing tree (*WPtree*) varies with the query load and the evaluation order of the predicates. The query predicates of query Q5 are *o_orderdate*, *r_name*, and *c_nationkey* and *s_nationkey*, which can originate a *WPtree* with up to 60 branches. Predicates can be arranged in 3 distinct *WPtrees*, based on the order they are evaluated: *c_nationkey* = *s_nationkey* → *o_orderdate* → *r_name* (*WPtree1*);

o_orderdate → *c_nationkey* = *s_nationkey* → *r_name* (*WPtree2*);

o_orderdate → *r_name* → *c_nationkey* = *s_nationkey* (*WPtree3*).

As the relation is spinning, and tuples are evaluated, several *bitsets* were built: 10 for the *o_orderdate* attribute, one *bitset* for each year, with a selectivity of roughly 10%, 5 *bitsets* for *r_name*, one for each region name, each with a selectivity of 20%, and also a *bitset* that represents the *nationkey* equality (*c_nationkey* = *s_nationkey*), where a bit is set to true when the tuple that satisfy the equality. This *bitset* has a selectivity of about 4%. We then analyzed the time required to evaluate the predicates. The results are depicted in Fig. 8.28. Since SPIN merges queries with matching predicates, the x-axis denotes the number of concurrent clients with distinct query predicates.

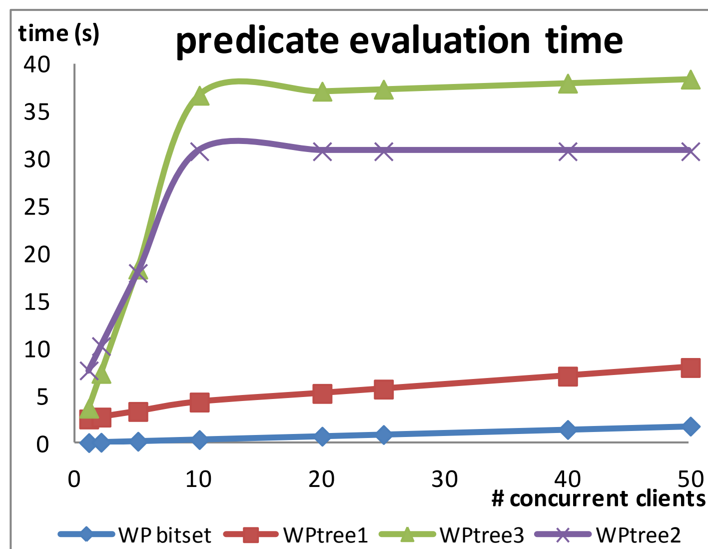


Fig. 8.28 – Total predicate evaluation time

From the figure we observe that the processing trees *WPtree2* and *WPtree3* have higher evaluation times, since the predicate evaluation order do not follow the predicate selectivity. *WPtree1* is the one that presents lower times, since the predicates are evaluated in an increasing order of selectivity. *Bitset* processing yields improved evaluation times, represented in the figure as *WP bitset*, since for each submitted query the predicates where replaced with a *bitset* lookup. The depicted result accounts for the time needed to parse and find the set of bitsets that match each of the query predicates, the time needed to build a composed *bitset* (perform an AND to the set of matching bitsets) and also the tuple evaluations using this *bitset*. For the considered query load, each query will have a composed *bitset*, which is obtained by combining bitsets that match the query predicates, with low selectivity ($10\% \times 20\% \times 4\%$).

The order of predicates and how the matching branches are orchestrated in the *WPtree* influences the predicate evaluation times. For large query loads, with distinct query patterns, it may not be possible to obtain the best order for each query. *Bitset* processing speedups the evaluation of predicates and avoid these issues.

Another feature of *bitset* processing is that we also reduce the number of tuple evaluations made by selection operators. When the current *WPtree* does do not require all the data, then the *WPbitset* can be pushed forward to the Data Reader to avoid reading unnecessary data fragments. For a query, Fig. 8.29 depicts the total number of tuple evaluations made by the selection operators (***σ***) with the base SPIN setup (SPIN) using *WPtree1*, when the *bitset* selection operators (***σbit***) are placed along the branches (SPIN-WP bitset), and when the Data Reader uses a *WPbitset* (SPIN-DR bitset). The depicted results consider that data is partitioned by *o_orderdate*.

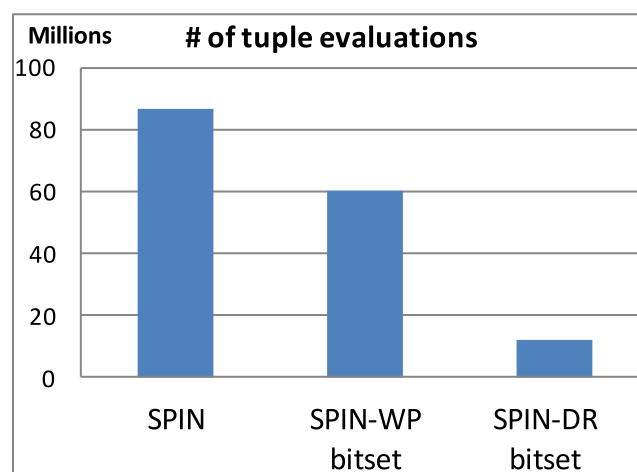


Fig. 8.29 – Number of tuple evaluations

Since SPIN-DR *bitset* early discards uninteresting tuples, subsequent tuple evaluations are performed only to relevant tuples. The number of tuple evaluations when SPIN-WP *bitset* is also significantly smaller than the base SPIN, since after individual *bitsets* have been created they can be combined according to the query predicates, resulting into a single *bitset* with low selectivity (10% × 20% × 4%). In this case the number of tuple evaluations is equal to the number of tuples, and thus it's particularly interesting to use a *WPbitset* to reduce the number of tuples that are read and placed in the base pipeline.

Next we evaluated the impact of *bitset* processing in the average execution time, for a varying number of concurrent clients. The results are depicted in Fig. 8.30.

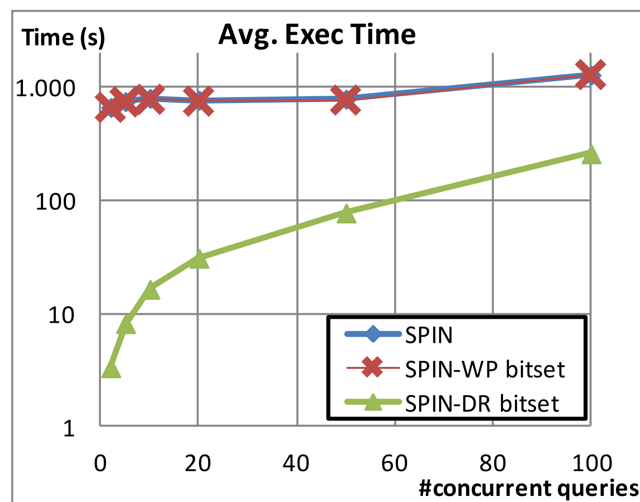


Fig. 8.30 – Average execution time

The results shows that even though *SPIN-WP bitset* evaluation is faster than tuple evaluation of SPIN (recall Fig. 8.28), *SPIN-WP bitset* only yields a slightly improvement in the average execution time in comparison with SPIN (in the graph they are almost the same). This is because we used a query load composed with variants of Q5, and SPIN setup uses the most efficient deployment (*WPtree1*), organized in utmost 60 branches, and also because the overall execution costs is constrained by the IO reading time. However when we apply the *WPbitset* to the data reader (SPIN-DR bitset) the performance is significantly better since it can avoid reading large number of tuples, and consequently reduce the number of evaluations and the evaluation time.

8.7 Chapter Summary

This chapter evaluated the approaches and mechanisms proposed in this dissertation.

The ONE data model and the query processing mechanisms offer reliable and predictable execution times, which can be estimated as a function of the data volume and the underlying storage system. As the execution time of the ONE data model increases linearly with the volume of data, DBAs and IT managers can estimate, with an appreciable confidence, how the supporting infrastructure will behave for larger data volumes. Moreover, since a large amount of the query execution cost is from IO operations, particularly the sustained transfer read rate, we can, with high confidence, estimate how the current hardware systems behave and the performance gains that can be obtained by upgrading the processing infrastructure, even without testing it.

The experimental results show that ONE-P scales-out almost linearly, and can be massively partitioned and distributed among an elastic set of processing nodes, and is able to provide timely results.

The experimental results also have shown that for large concurrent query loads, the SPIN processing model is capable of sharing data and processing in order to provide efficient and predictable execution times.

Chapter 9

Conclusions and Future Work

This chapter presents the conclusions of this research and gives some directions of future work that is worth of further investigation.

9.1 Conclusions

This dissertation proposes mechanisms for handling massively scalable data warehouses with performance predictability. We have shown that the scalability and the ability to provide timely results are constrained by the data and processing model.

We proposed an alternative data model, called ONE [Costa, Cecílio, et al. 2011; Costa, Martins, et al. 2011], which de-normalizes the star schema and thus trades storage space with the ability to provide predictable execution times, even for larger data volumes. The data model has reduced memory requirements and is IO dependent, i.e., storage throughput is the factor that most influences execution time. However, it can be massively partitioned among a wide set of processing nodes with almost linear speedup. We discussed the storage requirements of ONE, and experimentally evaluated its performance, scalability and predictability features.

Parallel shared-nothing infrastructures are usually used to handle large DWs. To overcome the scalability limitations of large parallel star schema DWs, and since the ONE data model provides predictable execution times for scalable data volumes, we have proposed ONE-P [Costa, Martins, et al. 2011; Costa, Cecílio, et al. 2012] which partitions the de-normalized relation into a set of data fragments and distributes them among processing nodes of the parallel infrastructure. The query execution time of ONE-P is predictable and can be estimated as a function of the number of nodes, and

the data volume allocated to each of the processing nodes. We experimentally demonstrated that ONE-P does not have the scalability limitations of the star schema model, and it scales-out and can deliver almost linear speedup without increasing the overall data storage size.

In large DWs, it is important to have high throughput, but it is also important to deliver timely results to queries (including ad-hoc queries). The ability to provide timely results before business decisions are made is gaining increasing interest. However the continuous increase in data volume and the need for faster results may endanger the ability of the parallel infrastructure to guarantee timely results. Therefore, to achieve that goal we proposed TEEPA [Costa, Martins, et al. 2012a; Costa, Martins, et al. 2012b], a Timely-aware Execution Parallel Architecture which balances the data volume allocated to each node, the data storage organization and the query processing among an elastic set of heterogeneous nodes to provide scale-out performance and timely query results. In each node, the data volume is stored using an adaptable storage model, which varyies the level of de-normalization that best fit the node' capabilities, in order to minimize the join costs (the major uncertainty factor), while preserving a consistent logical view of the star schema. The experimental results demonstrated that TEEPA is able to adapt the underlying infrastructure and storage organization to provide timely results.

We also proposed SPIN [Costa & Furtado 2013; Costa & Furtado 2015], a data and processing sharing model that delivers predictable execution times to aggregation queries, even in the presence of large concurrent workloads, without the memory and scalability limitations of existing approaches. For the current query load it combines and merges similar predicates and computation into common pipelines and then avoids the concurrent execution of similar tasks. The remaining query processing tasks, which cannot be combined with others, are plugged as branches of these common pipelines, creating a workload processing tree. Experiments have proven the usefulness of the proposed approach, and have shown that SPIN is able to provide scalable performance and predictable execution times even in presence of large concurrent query loads. To handle large query loads, we extended SPIN and proposed a *bitset* processing approach that builds a *bitset* for each branch, with the branch results for each of the evaluated tuples. Since SPIN processes data in circular fashion, a branch that processes data which was evaluated in previous loops can be replaced by faster *bitset* lookup. Along a

query path, a set of consecutive *bitsets* can be combined into a merged *bitset*, therefore reducing even further the cost of processing the workload processing tree.

When the query execution time is higher than a query time target, we proposed a parallel SPIN approach, named CARROUSEL, which manages a set of SPIN processing engines in parallel, to reduce the query execution time below that time target. Besides the data volume, it redistributes queries, query processing and data branches, among processing nodes, according to query load and data fragments that each node stores.

9.2 Future Work

We are currently pursuing a set of research directions that were not included in this dissertation, which have the potential to provide further performance improvements. The ONE cost models and the experimental evaluation was conducted using the typical row-store organization of most DBMS engines. However, columnar-store organization has proven to provide enhanced performance gains, since only the required columns have to be read from storage. A columnar ONE model has the scalability characteristics of a single relation, with a simpler and no-join approach where columns are combined based on tuple position, with a significant reduction in data size. A columnar approach will also yield performance gains to SPIN processing model, with different data readers (for distinct columns) being attached to specific branches. CARROUSEL will also benefit from it, since only the columns required by the balanced branches have to be rearranged (copied) among processing nodes.

Data compression is another research direction that we are currently pursuing, in order to reduce the storage requirements and consequently the time required to read the de-normalized data. The deployment of large DW over a Hadoop infrastructure with the scalability of ONE data model is a direction that should be explored.

Regarding the SPIN processing model, we are also considering the possibility to trade precision for performance as an alternative approach to provide timely results.

References

[Abiteboul, Duschka1998] "*Complexity of Answering Queries Using Materialized Views,*" Abiteboul, Serge, & Oliver M. Duschka, 1998. In *Proceedings of the 17th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, 254–263. PODS '98. New York, NY, USA: ACM.

[Abouzeid, Bajda-Pawlikowski, et al.2009] "*HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads,*" Abouzeid, Azza, Kamil Bajda-Pawlikowski, Daniel Abadi, Avi Silberschatz, & Alexander Rasin, 2009. *Proceedings of the VLDB Endowment* 2 (1) (August): 922–933.

[Acharya, Gibbons, et al.2000] "*Congressional Samples for Approximate Answering of Group-by Queries,*" Acharya, Swarup, Phillip B. Gibbons, & Viswanath Poosala, 2000. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, 487–498. SIGMOD '00. New York, NY, USA: ACM.

[Acharya, Gibbons, et al.1999] "*Join Synopses for Approximate Query Answering,*" Acharya, Swarup, Phillip B. Gibbons, Viswanath Poosala, & Sridhar Ramaswamy, 1999. *SIGMOD Rec.* 28 (2) (June): 275–286.

[Achyutuni, Omiecinski, et al.1995] *The Impact of Data Placement Strategies on Reorganization Costs in Parallel Databases*, Achyutuni, Kiran J., Edward Omiecinski, & Shamkant B. Navathe, 1995. .

[Agrawal, Chaudhuri, et al.2000] "*Automated Selection of Materialized Views and Indexes in SQL Databases,*" Agrawal, Sanjay, Surajit Chaudhuri, & Vivek R. Narasayya, 2000. In *Proceedings of the 26th International Conference on Very Large Data Bases*, 496–505. VLDB '00. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

[Arumugam, Dobra, et al.2010] "*The DataPath System: A Data-Centric Analytic Processing Engine for Large Data Warehouses,*" Arumugam, Subi, Alin Dobra, Christopher M Jermaine, Niketan Pansare, & Luis Perez, 2010. *Proceedings of the 2010 International Conference on Management of Data*. SIGMOD '10: 519–530.

[Baru, Fecteau, et al.1995] "*DB2 Parallel Edition,*" Baru, C.K., G. Fecteau, A. Goyal, H. Hsiao, A. Jhingran, S. Padmanabhan, G.P. Copeland, & W.G. Wilson, 1995. *IBM Systems Journal* 34 (2): 292–322.

[Bayer, McCreight1970] "*Organization and Maintenance of Large Ordered Indices,*" Bayer, R., & E. McCreight, 1970. In *Proceedings of the ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control*, 107–141. SIGFIDET '70. New York, USA: ACM.

[Bellatreche, Karlapalem, et al.2000] "*What Can Partitioning Do for Your Data Warehouses and Data Marts?,*" Bellatreche, L., K. Karlapalem, M. Mohania, & M. Schneider, 2000. In *Database Engineering and Applications Symposium, 2000 International*, 437–445.

[Bernardino2002] "*Técnicas para o aumento do desempenho e da disponibilidade em data warehouses,*" Bernardino, Jorge Fernandes Rodrigues, 2002. (January 24).

[Blasgen, Eswaran1977] "*Storage and Access in Relational Data Bases,*" Blasgen, M W, & K. P. Eswaran, 1977. *IBM Systems Journal* 16 (4): 363–377.

[Candea, Polyzotis, et al.2009] "*A Scalable, Predictable Join Operator for Highly Concurrent Data Warehouses,*" Candea, George, Neoklis Polyzotis, & Radek Vingralek, 2009. *Proceedings of the VLDB Endowment* 2 (August): 277–288.

[Candea, Polyzotis, et al.2011] "*Predictable Performance and High Query Concurrency for Data Analytics,*" Candea, George, Neoklis Polyzotis, & Radek Vingralek, 2011. *The VLDB Journal* 20 (2) (April): 227–248.

[Chan, Ioannidis1998] "*Bitmap Index Design and Evaluation,*" Chan, Chee-Yong, & Yannis E. Ioannidis, 1998. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, 355–366. SIGMOD '98. New York, NY, USA: ACM.

[Chaudhuri, Dayal1997] "*An Overview of Data Warehousing and OLAP Technology,*" Chaudhuri, Surajit, & Umeshwar Dayal, 1997. *SIGMOD Rec.* 26 (1): 65–74.

[Chen, Gehrke, et al.2001] "*Query Optimization In Compressed Database Systems,*" Chen, Zhiyuan, Johannes Gehrke, & Flip Korn, 2001. In *In ACM SIGMOD*, 271–282. ACM Press.

[Cheung, Madden, et al.2012] "*Automatic Partitioning of Database Applications,*" Cheung, Alvin, Samuel Madden, Owen Arden, & Andrew C. Myers, 2012. *Proc. VLDB Endow.* 5 (11) (July): 1471–1482.

[Comer1979] "*Ubiquitous B-Tree,*" Comer, Douglas, 1979. *ACM Comput. Surv.* 11 (2) (June): 121–137.

[Copeland, Alexander, et al.1988] "*Data Placement in Bubba,*" Copeland, George, William Alexander, Ellen Boughter, & Tom Keller, 1988. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*.

[Copeland, Khoshafian1985] "*A Decomposition Storage Model,*" Copeland, George P, & Setrag N Khoshafian, 1985. *Proceedings of the ACM-SIGMOD International Conference on Management of Data*.

[Costa, Cecílio, et al.2011] "*ONE: A Predictable and Scalable DW Model,*" Costa, João Pedro, José Cecílio, Pedro Martins, & Pedro Furtado, 2011. In *Proceedings of the 13th International Conference on Data Warehousing and Knowledge Discovery*, 1–13. DaWaK'11. Toulouse, France: Springer-Verlag.

[Costa, Cecílio, et al.2012] "*Overcoming the Scalability Limitations of Parallel Star Schema Data Warehouses,*" Costa, João Pedro, José Cecílio, Pedro Martins, & Pedro Furtado, 2012. In *Proceedings of the 12th International Conference on Algorithms and Architectures for Parallel Processing - Volume Part I*, 473–486. ICA3PP'12. Berlin, Heidelberg: Springer-Verlag.

[Costa, Furtado2003] "*Time-Stratified Sampling for Approximate Answers to Aggregate Queries,*" Costa, João Pedro, & Pedro Furtado, 2003. In *International Conference on Database Systems for Advanced Applications (DASFAA 2003)*, 0:215. Kyoto, Japan: IEEE Computer Society.

[Costa, Furtado2013] "*SPIN:Concurrent Workload Scaling over Data Warehouses,*" Costa, João, & Pedro Furtado, 2013. In *Proceedings of the 15th International*

Conference on Data Warehousing and Knowledge Discovery - DaWaK 2013. Prague, Czech Republic.

[Costa, Furtado2015] "*Data Warehouse Processing Scale-Up for Massive Concurrent Queries with SPIN,*" Costa, João, & Pedro Furtado, 2015. In *Transactions on Large-Scale Data- and Knowledge-Centered Systems XVII*, ed by. Abdelkader Hameurlain, Josef Küng, Roland Wagner, Ladjel Bellatreche, & Mukesh Mohania, 1–23. Lecture Notes in Computer Science 8970. Springer Berlin Heidelberg.

[Costa, Madeira2004] "*Handling Big Dimensions in Distributed Data Warehouses Using the DWS Technique,*" Costa, Marco, & Henrique Madeira, 2004. In *Proceedings of the 7th ACM International Workshop on Data Warehousing and OLAP*, 31–37. DOLAP '04. New York, NY, USA: ACM.

[Costa, Martins, et al.2011] "*A Predictable Storage Model for Scalable Parallel DW,*" Costa, João Pedro, Pedro Martins, José Cecílio, & Pedro Furtado, 2011. In *Fifteenth International Database Engineering and Applications Symposium (IDEAS 2011)*. Lisbon, Portugal: ACM.

[Costa, Martins, et al.2012a] "*TEEPA: A Timely-Aware Elastic Parallel Architecture,*" Costa, João Pedro, Pedro Martins, José Cecilio, & Pedro Furtado, 2012. In *Proceedings of the 16th International Database Engineering & Applications Symposium*, 24–31. IDEAS'12. New York, NY, USA: ACM.

[Costa, Martins, et al.2012b] "*Providing Timely Results with an Elastic Parallel DW,*" Costa, João Pedro, Pedro Martins, José Cecilio, & Pedro Furtado, 2012. In *Proceedings of the 20th International Conference on Foundations of Intelligent Systems*, 415–424. ISMIS'12. Berlin, Heidelberg: Springer-Verlag.

[Dewitt, Ghandeharizadeh, et al.1990] "*The Gamma Database Machine Project,*" Dewitt, David J., Shahram Ghandeharizadeh, Donovan Schneider, Allan Bricker, Hui-i Hsiao, & Rick Rasmussen, 1990. *IEEE Transactions on Knowledge and Data Engineering* 2: 44–62.

[Dewitt, Gray1992] "*Parallel Database Systems: The Future of High Performance Database Systems,*" Dewitt, David J., & Jim Gray, 1992. *Communications of the ACM* 35: 85–98.

[DeWitt, Katz, et al.1984] "*Implementation Techniques for Main Memory Database Systems,*" DeWitt, David J, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, & David A Wood, 1984. In *ACM SIGMOD Record*, 1–8. SIGMOD '84. New York, NY, USA: ACM.

[DeWitt, Naughton, et al.1992] "*Practical Skew Handling in Parallel Joins,*" DeWitt, David J., Jeffrey F. Naughton, Donovan A. Schneider, & S. Seshadri, 1992. In *Proceedings of the 18th International Conference on Very Large Data Bases*, 27–40. VLDB '92. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

[Epstein, Stonebraker, et al.1978] "*Distributed Query Processing in a Relational Data Base System,*" Epstein, Robert, Michael Stonebraker, & Eugene Wong, 1978. In *Proceedings of the 1978 ACM SIGMOD International Conference on Management of Data*, 169–180. SIGMOD '78. New York, NY, USA: ACM.

[Fagin, Mendelzon, et al.1982] "*A Simplied Universal Relation Assumption and Its Properties,*" Fagin, Ronald, Alberto O. Mendelzon, & Jeffrey D. Ullman, 1982. *ACM Trans. Database Syst.* 7 (3) (September): 343–360.

[Frey, Goncalves, et al.2009] "*Spinning Relations: High-Speed Networks for Distributed Join Processing,*" Frey, Philip W, Romulo Goncalves, Martin Kersten, & Jens Teubner, 2009. *Proceedings of the Fifth International Workshop on Data Management on New Hardware*. DaMoN '09: 27–33.

[Frey, Goncalves, et al.2010] "*A Spinning Join That Does Not Get Dizzy,*" Frey, Philip W., Romulo Goncalves, Martin Kersten, & Jens Teubner, 2010. In *20th International Conference on Distributed Computing Systems (ICDCS 2010)*, 0:283–292. Los Alamitos, CA, USA: IEEE Computer Society.

[Furtado2004] "*Workload-Based Placement and Join Processing in Node-Partitioned Data Warehouses,*" Furtado, Pedro, 2004. In *Data Warehousing and Knowledge Discovery*, 3181:38–47. Lecture Notes in Computer Science. Springer Berlin / Heidelberg.

[Furtado2008] "*Efficient, Chunk-Replicated Node Partitioned Data Warehouses,*" Furtado, Pedro, 2008. In *2008 IEEE International Symposium on Parallel and Distributed Processing with Applications*, 578–583. Sydney, Australia.

[Furtado2009a] "*A Survey of Parallel and Distributed Data Warehouses,*" Furtado, Pedro, 2009. *IJDWM* 5 (2): 57–77.

[Furtado2009b] "*Model and Procedure for Performance and Availability-Wise Parallel Warehouses,*" Furtado, Pedro, 2009. *Distributed and Parallel Databases* 25 (1): 71–96.

[Giannikis, Alonso, et al.2012] "*SharedDB: Killing One Thousand Queries with One Stone,*" Giannikis, Georgios, Gustavo Alonso, & Donald Kossmann, 2012. *Proc. VLDB Endow.* 5 (6) (February): 526–537.

[Golfarelli, Rizzi2009] "*A Survey on Temporal Data Warehousing,*" Golfarelli, Matteo, & Stefano Rizzi, 2009. In *Database Technologies: Concepts, Methodologies, Tools, and Applications*, 221–237.

[Graefe1993] "*Query Evaluation Techniques for Large Databases,*" Graefe, Goetz, 1993. *ACM Comput. Surv.* 25 (2) (June): 73–169.

[Graefe2011] "*New Algorithms for Join and Grouping Operations,*" Graefe, Goetz, 2011. *Computer Science - Research and Development* (June).

[Graefe, Linville, et al.1994] "*Sort vs. Hash Revisited,*" Graefe, G., A. Linville, & L. D Shapiro, 1994. *IEEE Transactions on Knowledge and Data Engineering* 6 (December): 934–944.

[Grund, Krüger, et al.2010] "*HYRISE: A Main Memory Hybrid Storage Engine,*" Grund, Martin, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudre-Mauroux, & Samuel Madden, 2010. *Proc. VLDB Endow.* 4 (November): 105–116.

[Gupta2000] "*Selection and Maintenance of Views in a Data Warehouse,*" Gupta, Himanshu Satishkumar, 2000. . Stanford, CA, USA: Stanford University.

[Harizopoulos, Shkapenyuk, et al.2005] "*QPipe: A Simultaneously Pipelined Relational Query Engine,*" Harizopoulos, Stavros, Vladislav Shkapenyuk, & Anastassia

Ailamaki, 2005. *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’05: 383–394.

[Harris, Ramamohanarao1996] “*Join Algorithm Costs Revisited,*” Harris, Evan P, & Kotagiri Ramamohanarao, 1996. *The VLDB Journal — The International Journal on Very Large Data Bases* 5 (January): 064–084.

[Holloway, Raman, et al.2007] “*How to Barter Bits for Chronons: Compression and Bandwidth Trade Offs for Database Scans,*” Holloway, Allison L., Vijayshankar Raman, Garret Swart, & David J. DeWitt, 2007. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, 389–400. SIGMOD ’07. New York, NY, USA: ACM.

[Huang, Zhang, et al.2009] “*A Data Distribution Strategy for Scalable Main-Memory Database,*” Huang, Yunkui, YanSong Zhang, Xiaodong Ji, Zhanwei Wang, & Shan Wang, 2009. In *Advances in Web and Network Technologies, and Information Management*, ed by. Lei Chen, Chengfei Liu, Xiao Zhang, Shan Wang, Darijus Strasunskas, Stein L. Tomassen, Jinghai Rao, et al., 13–24. Lecture Notes in Computer Science 5731. Springer Berlin Heidelberg.

[Jaedicke, Mitschang1998] “*On Parallel Processing of Aggregate and Scalar Functions in Object-Relational DBMS,*” Jaedicke, Michael, & Bernhard Mitschang, 1998. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, 379–389. SIGMOD ’98. New York, NY, USA: ACM.

[Jermaine, Pol, et al.2004] “*Online Maintenance of Very Large Random Samples,*” Jermaine, Christopher, Abhijit Pol, & Subramanian Arumugam, 2004. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, 299–310. SIGMOD ’04. New York, NY, USA: ACM.

[Johnson1999] “*Performance Measurements of Compressed Bitmap Indices,*” Johnson, Theodore, 1999. *Proceedings of the 25th International Conference on Very Large Data Bases*. VLDB ’99: 278–289.

[Kallman, Kimura, et al.2008] “*H-Store: A High-Performance, Distributed Main Memory Transaction Processing System,*” Kallman, Robert, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, et al., 2008. *Proc. VLDB Endow.* 1 (2): 1496–1499.

[Kimball1996] *The Data Warehouse Toolkit: Practical Techniques for Building Dimensional Data Warehouses*, Kimball, Ralph, 1996. . Wiley.

[Kimball, Ross, et al.2008] *The Data Warehouse Lifecycle Toolkit*, Kimball, Ralph, Margy Ross, Warren Thornthwaite, Joy Mundy, & Bob Becker, 2008. . 2nd ed. Wiley Publishing.

[Kim, Kaldewey, et al.2009] “*Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs,*” Kim, Changkyu, Tim Kaldewey, Victor W Lee, Eric Sedlar, Anthony D Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, & Pradeep Dubey, 2009. *Proceedings of the VLDB Endowment* 2 (2) (August): 1378–1389.

[Kitsuregawa, Tanaka, et al.1983] “*Application of Hash to Data Base Machine and Its Architecture,*” Kitsuregawa, Masaru, Hidehiko Tanaka, & Tohru Moto-Oka, 1983. *New Generation Computing* 1 (1): 63–74.

[Korth, Kuper, et al.1984] "*SYSTEM/U: A Database System Based on the Universal Relation Assumption,*" Korth, Henry F., Gabriel M. Kuper, Joan Feigenbaum, Allen van Gelder, & Jeffrey D. Ullman, 1984. *ACM Trans. Database Syst.* 9 (3) (September): 331–347.

[Lemire, Kaser2011] "*Reordering Columns for Smaller Indexes,*" Lemire, Daniel, & Owen Kaser, 2011. *Inf. Sci.* 181 (12) (June): 2550–2570.

[Lemire, Kaser, et al.2012] "*Reordering Rows for Better Compression: Beyond the Lexicographic Order,*" Lemire, Daniel, Owen Kaser, & Eduardo Gutarra, 2012. *ACM Trans. Database Syst.* 37 (3) (September): 20:1–20:29.

[Li, Patel2014] "*WideTable: An Accelerator for Analytical Data Processing,*" Li, Yinan, & Jignesh M. Patel, 2014. *PVLDB* 7 (10): 907–918.

[Liu, Chen1996] "*A Hash Partition Strategy for Distributed Query Processing,*" Liu, Chengwen, & Hao Chen, 1996. In *Advances in Database Technology - EDBT '96*, ed by. Peter Apers, Mokrane Bouzeghoub, & Georges Gardarin, 1057:371–387. Berlin/Heidelberg: Springer-Verlag.

[Liu, Yu1992] "*Validation and Performance Evaluation of the Partition and Replicate Algorithm,*" Liu, Chengwen, & C. Yu, 1992. In *, Proceedings of the 12th International Conference on Distributed Computing Systems, 1992*, 400–407.

[Mami, Bellahsene2012] "*A Survey of View Selection Methods,*" Mami, Imene, & Zohra Bellahsene, 2012. *SIGMOD Rec.* 41 (1) (April): 20–29.

[Mehta, DeWitt1997] "*Data Placement in Shared-Nothing Parallel Database Systems,*" Mehta, Manish, & David J. DeWitt, 1997. *The VLDB Journal* 6 (1) (February): 53–72.

[Mishra, Eich1992] "*Join Processing in Relational Databases,*" Mishra, Priti, & Margaret H. Eich, 1992. *ACM Computing Surveys* 24: 63–113.

[Noaman, Barker1999] "*A Horizontal Fragmentation Algorithm for the Fact Relation in a Distributed Data Warehouse,*" Noaman, Amin Y., & Ken Barker, 1999. In *Proceedings of the Eighth International Conference on Information and Knowledge Management*, 154–161. CIKM '99. New York, NY, USA: ACM.

[O'Connell, Winterbottom2003] "*Performing Joins without Decompression in a Compressed Database System,*" O'Connell, S. J., & N. Winterbottom, 2003. *SIGMOD Rec.* 32 (1) (March): 6–11.

[O'Neil, Graefe1995] "*Multi-Table Joins through Bitmapped Join Indices,*" O'Neil, Patrick, & Goetz Graefe, 1995. *ACM SIGMOD Record* 24 (3) (September): 8–11.

[O'Neil, O'Neil, et al.2009] "*The Star Schema Benchmark and Augmented Fact Table Indexing,*" O'Neil, Patrick, Elizabeth O'Neil, Xuedong Chen, & Stephen Revilak, 2009. In *Performance Evaluation and Benchmarking*, 5895:237–252. Lecture Notes in Computer Science. Springer Berlin / Heidelberg.

[O'Neil, O'Neil, et al.2007] "*Bitmap Index Design Choices and Their Performance Implications,*" O'Neil, E., P. O'Neil, & Kesheng Wu, 2007. In *Database Engineering and Applications Symposium, 2007. IDEAS 2007. 11th International*, 72–84.

[O'Neil, Quass1997] "*Improved Query Performance with Variant Indexes,*" O'Neil, Patrick, & Dallan Quass, 1997. *SIGMOD Rec.* 26 (2) (June): 38–49.

[Ozsu, Valduriez2011] *Principles of Distributed Database Systems*, Ozsu, M. Tamer, & Patrick Valduriez, 2011. . 3rd ed. Springer Publishing Company, Incorporated.

[Patel, Carey, et al.1994] "*Accurate Modeling of the Hybrid Hash Join Algorithm,*" Patel, Jignesh M, Michael J Carey, & Mary K Vernon, 1994. In *ACM SIGMETRICS Performance Evaluation Review*. SIGMETRICS '94. NY, USA: ACM.

[Patterson, Ditzel1980] "*The Case for the Reduced Instruction Set Computer,*" Patterson, David A., & David R. Ditzel, 1980. *SIGARCH Comput. Archit. News* 8 (6) (October): 25–33.

[Pavlo, Curino, et al.2012] "*Skew-Aware Automatic Database Partitioning in Shared-Nothing, Parallel OLTP Systems,*" Pavlo, Andrew, Carlo Curino, & Stanley Zdonik, 2012. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 61–72. SIGMOD '12. New York, NY, USA: ACM.

[Pavlo, Paulson, et al.2009] "*A Comparison of Approaches to Large-Scale Data Analysis,*" Pavlo, Andrew, Erik Paulson, Alexander Rasin, Daniel J Abadi, David J DeWitt, Samuel Madden, & Michael Stonebraker, 2009. *Proc. of the 35th SIGMOD International Conference on Management of Data*. SIGMOD '09: 165–178.

[Pinto2009] "*A Framework for Systematic Database Denormalization,*" Pinto, Yma, 2009. *Global Journal of Computer Science and Technology* 9 (4) (August 15).

[Poess, Potapov2003] "*Data Compression in Oracle,*" Poess, Meikel, & Dmitry Potapov, 2003. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, 937–947. VLDB '03. Berlin, Germany: VLDB Endowment.

[Raman, Swart, et al.2008] "*Constant-Time Query Processing,*" Raman, Vijayshankar, Garret Swart, Lin Qiao, Frederick Reiss, Vijay Dialani, Donald Kossmann, Inderpal Narang, & Richard Sidle, 2008. *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*: 60–69.

[Roussopoulos1998] "*Materialized Views and Data Warehouses,*" Roussopoulos, Nick, 1998. *SIGMOD Rec.* 27 (1) (March): 21–26.

[Sanders, Shin2001] "*Denormalization Effects on Performance of RDBMS,*" Sanders, G. Lawrence, & Seungkyoon Shin, 2001. In *Proceedings of the 34th Annual Hawaii International Conference on System Sciences*, 3:3013. HICSS '01. Washington, DC, USA: IEEE Computer Society.

[Schneider, Dewitt1989] "*A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment,*" Schneider, Donovan A, & David J Dewitt, 1989. : 110–121.

[Shasha, Wang1991] "*Optimizing Equijoin Queries in Distributed Databases Where Relations Are Hash Partitioned,*" Shasha, Dennis, & Tsong-Li Wang, 1991. *ACM Transactions on Database Systems* 16 (2) (May): 279–308.

[Shatdal, Naughton1994] *Processing Aggregates in Parallel Database Systems*, Shatdal, Ambuj, & Jeffrey F. Naughton, 1994. . Vol. Computer Sciences Technical Report #123. University of Wisconsin-Madison, Computer Sciences Department.

[Stockinger, Wu2006] "*Bitmap Indices for Data Warehouses,*" Stockinger, Kurt, & Kesheng Wu, 2006. In *In Data Warehouses and OLAP. 2007. IRM*. Press.

[Stöhr, Märtens, et al.2000] "*Multi-Dimensional Database Allocation for Parallel Data Warehouses,*" Stöhr, Thomas, Holger Märtens, & Erhard Rahm, 2000. In *Proceedings of the 26th International Conference on Very Large Data Bases*, 273–284. VLDB '00. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

[Stonebraker, Abadi, et al.2005] "*C-Store: A Column-Oriented DBMS,*" Stonebraker, Mike, Daniel J Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, et al., 2005. *Proceedings of the 31st International Conference on Very Large Data Bases*. VLDB '05: 553–564.

[Strohm2011] "*Oracle Database Concepts, 11g Release 1 (11.1),*" Strohm, Richard, 2011. *Oracle Database Concepts, 11g Release 1 (11.1).*

[Thusoo, Sarma, et al.2009] "*Hive: A Warehousing Solution over a Map-Reduce Framework,*" Thusoo, Ashish, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, & Raghotham Murthy, 2009. *Proc. VLDB Endow.* 2 (2): 1626–1629.

[TPC-H] "*TPC-H Decision Support Benchmark,*" TPC-H, .

[Unterbrunner, Giannikis, et al.2009] "*Predictable Performance for Unpredictable Workloads,*" Unterbrunner, P., G. Giannikis, G. Alonso, D. Fauser, & D. Kossmann, 2009. *Proceedings of the VLDB Endowment* 2 (August): 706–717.

[Westmann, Kossmann, et al.1998] *The Implementation and Performance of Compressed Databases*, Westmann, Till, Donald Kossmann, Sven Helmer, & Guido Moerkotte, 1998. .

[Westmann, Kossmann, et al.2000] "*The Implementation and Performance of Compressed Databases,*" Westmann, Till, Donald Kossmann, Sven Helmer, & Guido Moerkotte, 2000. *SIGMOD Rec.* 29 (3) (September): 55–67.

[Wu, Madden2011] "*Partitioning Techniques for Fine-Grained Indexing,*" Wu, Eugene, & Samuel Madden, 2011. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, 1127–1138. ICDE '11. Washington, DC, USA: IEEE Computer Society.

[Xu, Kostamaa2009] "*Efficient Outer Join Data Skew Handling in Parallel DBMS,*" Xu, Yu, & Pekka Kostamaa, 2009. *Proc. VLDB Endow.* 2 (2) (August): 1390–1396.

[Xu, Kostamaa, et al.2010] "*Integrating Hadoop and Parallel DBMs,*" Xu, Yu, Pekka Kostamaa, & Like Gao, 2010. In *Proceedings of the 2010 International Conference on Management of Data*, 969–974. SIGMOD '10. New York, NY, USA: ACM.

[Xu, Kostamaa, et al.2008] "*Handling Data Skew in Parallel Joins in Shared-Nothing Systems,*" Xu, Yu, Pekka Kostamaa, Xin Zhou, & Liang Chen, 2008. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, 1043–1052. SIGMOD '08. New York, NY, USA: ACM.

[Yang, Yen, et al.2010] "*Osprey: Implementing MapReduce-Style Fault Tolerance in a Shared-Nothing Distributed Database,*" Yang, Christopher, Christine Yen,

Ceryen Tan, & Samuel R. Madden, 2010. In *Data Engineering, International Conference on*, 0:657–668. Los Alamitos, CA, USA: IEEE Computer Society.

[Yu, Guh, et al.1989] "*Partition Strategy for Distributed Query Processing in Fast Local Networks,*" Yu, Clement T., Keh-Chang Guh, David Brill, & Arbee L. P. Chen, 1989. *IEEE Trans. Softw. Eng.* 15 (6) (June): 780–793.

[Zaker, Phon-Amnuaisuk, et al.2008] "*Optimizing the Data Warehouse Design by Hierarchical Denormalizing,*" Zaker, Morteza, Somnuk Phon-Amnuaisuk, & Su-Cheng Haw, 2008. *Proc. 8th Conference on Applied Computer Science*.

[Zaker, Phon-amnuaisuk, et al.2009] "*Hierarchical Denormalizing: A Possibility to Optimize the Data Warehouse Design,*" Zaker, Morteza, Somnuk Phon-amnuaisuk, & Su-cheng Haw, 2009. .

[Zeller, Gray1990] "*An Adaptive Hash Join Algorithm for Multiuser Environments,*" Zeller, Hansjo"rrg, & Jim Gray, 1990. In *Proceedings of the 16th International Conference on Very Large DataBases*, 186–197. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

[Zhang, Hu, et al.2010] "*MOSS-DB: A Hardware-Aware OLAP Database,*" Zhang, Yansong, Wei Hu, & Shan Wang, 2010. In *Proceedings of the 11th International Conference on Web-Age Information Management*, 582–594. WAIM'10. Berlin, Heidelberg: Springer-Verlag.

[Zhou, Larson, et al.2007] "*Dynamic Materialized Views,*" Zhou, Jingren, Per-Ake Larson, Jonathan Goldstein, & Luping Ding, 2007. In *IEEE 23rd International Conference on Data Engineering*, 526–535. Los Alamitos, CA, USA: IEEE Computer Society.

[Zukowski, Héman, et al.2007] "*Cooperative Scans: Dynamic Bandwidth Sharing in a DBMS,*" Zukowski, Marcin, Sándor Héman, Niels Nes, & Peter Boncz, 2007. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, 723–734. VLDB '07. Vienna, Austria: VLDB Endowment.

Appendix A – TPC-H Queries

```
-- TPC-H/TPC-R Pricing Summary Report Query (Q1)
-- Functional Query Definition
-- Approved February 1998
SELECT l_returnflag, l_linestatus,
  sum(l_quantity) as sum_qty,
  sum(l_extendedprice) as sum_base_price,
  sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
  sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge,
  avg(l_quantity) as avg_qty,
  avg(l_extendedprice) as avg_price,
  avg(l_discount) as avg_disc,
  count(*) as count_order
FROM  lineitem
WHERE  l_shipdate <= date '1998-12-01' - interval ':1' day (3)
group by
  l_returnflag, l_linestatus
order by
  l_returnflag, l_linestatus;
```

```
-- TPC-H/TPC-R Minimum Cost Supplier Query (Q2)
-- Functional Query Definition
-- Approved February 1998
SELECT s_acctbal, s_name, n_name, p_partkey, p_mfgr,
       s_address, s_phone, s_comment
FROM   part, supplier, partsupp, nation, region
WHERE  p_partkey = ps_partkey
    and s_suppkey = ps_suppkey
    and p_size = :1
    and p_type like '%:2'
    and s_nationkey = n_nationkey
    and n_regionkey = r_regionkey
    and r_name = ':3'
    and ps_supplycost = (
        SELECT min(ps_supplycost)
        FROM
            partsupp, supplier, nation, region
        WHERE
            p_partkey = ps_partkey
            and s_suppkey = ps_suppkey
            and s_nationkey = n_nationkey
            and n_regionkey = r_regionkey
            and r_name = ':3'
    )
ORDER BY
    s_acctbal desc, n_name,   s_name, p_partkey;
```

```
-- TPC-H/TPC-R Shipping Priority Query (Q3)
-- Functional Query Definition
-- Approved February 1998
SELECT l_orderkey, sum(l_extendedprice * (1 - l_discount)) as revenue,
       o_orderdate, o_shippriority
FROM   customer, orders, lineitem
WHERE  c_mktsegment = ':1'
   and c_custkey = o_custkey
   and l_orderkey = o_orderkey
   and o_orderdate < date ':2'
   and l_shipdate > date ':2'
GROUP BY l_orderkey, o_orderdate, o_shippriority
ORDER BY revenue desc, o_orderdate;
```

```
-- TPC-H/TPC-R Order Priority Checking Query (Q4)
-- Functional Query Definition
-- Approved February 1998
SELECT o_orderpriority,      count(*) as order_count
FROM   orders
WHERE  o_orderdate >= date ':1'
   and o_orderdate < date ':1' + interval '3' month
   and exists (
       SELECT *
       FROM lineitem
       WHERE  l_orderkey = o_orderkey
          and l_commitdate < l_receiptdate
   )
GROUP BY o_orderpriority
ORDER BY o_orderpriority;
```

```
-- TPC-H/TPC-R Local Supplier Volume Query (Q5)
-- Functional Query Definition
-- Approved February 1998
SELECT n_name, sum(l_extendedprice * (1 - l_discount)) as revenue
FROM   customer, orders, lineitem, supplier, nation, region
WHERE  c_custkey = o_custkey
   and l_orderkey = o_orderkey
   and l_suppkey = s_suppkey
   and c_nationkey = s_nationkey
   and s_nationkey = n_nationkey
   and n_regionkey = r_regionkey
   and r_name = ':1'
   and o_orderdate >= date ':2'
   and o_orderdate < date ':2' + interval '1' year
GROUP BY n_name
ORDER BY revenue desc;
```

```
-- TPC-H/TPC-R Forecasting Revenue Change Query (Q6)
-- Functional Query Definition
-- Approved February 1998
SELECT sum(l_extendedprice * l_discount) as revenue
FROM   lineitem
WHERE l_shipdate >= date ':1'
    and l_shipdate < date ':1' + interval '1' year
    and l_discount between :2 - 0.01 and :2 + 0.01
    and l_quantity < :3;

-- TPC-H/TPC-R Volume Shipping Query (Q7)
-- Functional Query Definition
-- Approved February 1998
SELECT supp_nation, cust_nation, l_year, sum(volume) as revenue
FROM   (
    SELECT n1.n_name as supp_nation,
        n2.n_name as cust_nation,
        extract(year from l_shipdate) as l_year,
        l_extendedprice * (1 - l_discount) as volume
    FROM   supplier, lineitem, orders, customer,
           nation n1, nation n2
    WHERE  s_suppkey = l_suppkey
        and o_orderkey = l_orderkey
        and c_custkey = o_custkey
        and s_nationkey = n1.n_nationkey
        and c_nationkey = n2.n_nationkey
        and (   (n1.n_name = ':1' and n2.n_name = ':2')
             or (n1.n_name = ':2' and n2.n_name = ':1') )
        and l_shipdate between date '1995-01-01' and date '1996-12-31'
    ) as shipping
GROUP BY supp_nation, cust_nation, l_year
ORDER BY supp_nation, cust_nation, l_year;

-- TPC-H/TPC-R National Market Share Query (Q8)
-- Functional Query Definition
-- Approved February 1998
SELECT o_year, sum(case when nation = ':1' then volume else 0 end) /
    sum(volume) as mkt_share
FROM   (
    SELECT extract(year from o_orderdate) as o_year,
        l_extendedprice * (1 - l_discount) as volume,
        n2.n_name as nation
    FROM   part, supplier, lineitem, orders, customer,
           nation n1, nation n2, region
    WHERE  p_partkey = l_partkey
        and s_suppkey = l_suppkey
        and l_orderkey = o_orderkey
        and o_custkey = c_custkey
        and c_nationkey = n1.n_nationkey
        and n1.n_regionkey = r_regionkey
        and r_name = ':2'
        and s_nationkey = n2.n_nationkey
        and o_orderdate between date '1995-01-01' and date '1996-12-31'
        and p_type = ':3'
    ) as all_nations
GROUP BY o_year
ORDER BY o_year;
```

```
-- TPC-H/TPC-R Product Type Profit Measure Query (Q9)
-- Functional Query Definition
-- Approved February 1998
SELECT nation, o_year, sum(amount) as sum_profit
FROM   (
    SELECT n_name as nation,
       extract(year from o_orderdate) as o_year,
    l_extendedprice*(1-l_discount)-ps_supplycost*l_quantity as amount
    FROM part, supplier, lineitem, partsupp, orders, nation
    WHERE   s_suppkey = l_suppkey
       and ps_suppkey = l_suppkey
       and ps_partkey = l_partkey
       and p_partkey = l_partkey
       and o_orderkey = l_orderkey
       and s_nationkey = n_nationkey
       and p_name like '%:1%'
    ) as profit
GROUP BY nation, o_year
ORDER BY nation, o_year desc;


-- TPC-H/TPC-R Returned Item Reporting Query (Q10)
-- Functional Query Definition
-- Approved February 1998
SELECT c_custkey, c_name,
    sum(l_extendedprice * (1 - l_discount)) as revenue,
    c_acctbal, n_name, c_address, c_phone, c_comment
FROM   customer, orders, lineitem, nation
WHERE  c_custkey = o_custkey
    and l_orderkey = o_orderkey
    and o_orderdate >= date ':1'
    and o_orderdate < date ':1' + interval '3' month
    and l_returnflag = 'R'
    and c_nationkey = n_nationkey
GROUP BY c_custkey, c_name, c_acctbal,   c_phone,
n_name, c_address, c_comment
ORDER BY revenue desc;
```