Fábio André Castanheira Luís Coelho **Towards a Transactional and Analytical Data Management System for Big Data**

UMinho | 2018

Universidade do Minho
Escola de Engenharia

Fábio André Castanheira Luís Coelho

Towards a Transactional and Analytical Data Management System for Big Data

The MAP-i Doctoral Programme in Informatics, of the Universities of Minho, Aveiro and Porto

universidade de aveiro

Universidade do Minho

U.PORTO

January 2018

Universidade do Minho
Escola de Engenharia

Fábio André Castanheira Luís Coelho

Towards a Transactional and Analytical Data Management System for Big Data

The MAP-i Doctoral Programme in Informatics, of the Universities of Minho, Aveiro and Porto

universidade de aveiro

Universidade do Minho

U.PORTO

supervisors:
Professor Doutor Rui Carlos Oliveira
Professor Doutor José Orlando Pereira

January 2018

DECLARAÇÃO

Nome: Fábio André Castanheira Luís Coelho

Endereço Electrónico: fabio.a.coelho@inesctec.pt   Telefone: 00351 963428949

Número do Bilhete de Identidade: 13456866

Título da Tese: Towards a transactional and analytical data management system for big data

Orientadores: Prof. Rui Carlos Mendes de Oliveira e

Prof. José Orlando Roque Nascimento Pereira

Ano de Conclusão: 2018

Designação do Doutoramento: The MAP-I Doctoral Program Of The Universities of Minho, Aveiro and Porto

Universidade do Minho, 24 de Janeiro de 2018

Assinatura: Fábio André Coelho.

STATEMENT OF INTEGRITY

I hereby declare having conducted my thesis with integrity. I confirm that I have not used plagiarism or any form of falsification of results in the process of the thesis elaboration.
I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

University of Minho, 24 Janeiro de 2018

Full name: FÁBIO ANDRE CASTANHEIRA LUIS COELHO

Signature: Fábio André Coelho.

Agradecimentos

O ano de 2011 marcou um ponto de viragem na minha vida quando, conjuntamente com 3 amigos, decidi rumar ao norte de Portugal. A cidade de Braga acolheu-me e foi no seio das fantásticas pessoas que por cá encontrei que vivi, cresci e me superei. Foi em particular no Grupo de Sistemas Distribuídos (GSD) da Universidade do Minho que esta jornada começou, durante os dois anos que durou o meu mestrado. Foi simultaneamente sorte e uma virtude ter encontrado um grupo de pessoas tão coeso, que profissionalmente sempre excedeu na vontade de entreajuda, e que a nível pessoal sempre apoiou nos momentos menos bons e festejou nos melhores. Afinal, tudo é mais fácil quando nos rodeamos de amigos. Foram sem dúvida alguma estes, os grandes fatores que me levaram a embarcar neste doutoramento, cujo culminar celebramos hoje, juntos. Muito obrigado e um bem haja a todos os que direta, ou indiretamente, participaram no percurso do meu doutoramento e que está patente neste documento.

Não posso deixar de remeter alguns cumprimentos particulares. Gostaria em primeiro lugar de agradecer aos meus orientadores. Ao Prof. Rui Carlos Oliveira, pela confiança que em mim depositou e por todas as experiências académicas e pessoais que durante todo este percurso me proporcionou. A sua motivação, profissionalismo e amizade excederam o que lhe era exigido e levaram-me a ser hoje a pessoa que sou, mais observador, mais confiante e pleno de motivação. Ao Prof. José Orlando Pereira, por todos os momentos em que me aconselhou, guiando a minha visão ao longo dos obstáculos que fomos encontrando. A sua contribuição no sucesso deste trabalho é por isso imensurável.

Não pode passar em vão o meu profundo agradecimento para com o Laboratório de Software Confiável, em particular para com o GSD. A todas as pessoas que o compõem e que por lá passaram durante os anos em que lá estive, nomeadamente: Ana Alonso, Catarina Leones, Filipe Campos, Francisco Cruz, Francisco Maia, Francisco Neves, João Paulo, Miguel Matos, Nuno Machado, Paula Rodrigues, Ricardo Gonçalves, Ricardo Macedo, Ricardo Vilaça, Rui Gonçalves, Rui Ribeiro, Rogério Pontes e demais corpo docente. A todos vós exprimo a minha profunda gratitude por proporcionarem o substrato fértil de onde colho hoje este trabalho, pois cada um de vós, na sua medida, contribuiu largamente para o sucesso desta etapa. *I would also like to thank Valerio Schiavoni from the Université de Neuchâtel for all the insights and joint work in the beggining of this PhD.*

Aos meus amigos de sempre, Eunice Tudela de Azevedo, Adriana Tudela de Azevedo e Bruno Miguel Agostinho, cuja amizade a distância não desvaneceu, solidificando-a a cada dia. Ao Diogo da Cunha Rodrigues, Samuel Santos Almeida e Vasco Miguel Coelho por serem os 3 com os quais esta jornada começou. Por todos os momentos de partilha, união e amizade. Por serem os irmãos que eu nunca tive. À Margarida Vasconcelos e ao Marcelo Dias. Por serem o casal mais formoso, unido e resiliente que jamais conheci. A vossa energia é espelho da nossa amizade. Por fim, a todos os amigos que fundei em Braga: Ana Capelo, Ana Carolina, Tatiana Conde, Carlos Silva e Rosa Mariana.

Os meus pais José Maria Coelho e Maria José Castanheira Luís Coelho são a base da minha família. É devido a eles, aos seus sacrifícios e à vontade que sempre mostraram em que estudasse que hoje atinjo este patamar. Este trabalho é para vós. Agradeço também aos meus tios e primos mais próximos.

Por último, agradeço à Ana que me tem acompanhado nestes últimos anos, e que contribui todos os dias para ser hoje a pessoa que sou. *Obrigado* :D.

Adicionalmente, algumas instituições apoiaram o presente trabalho. Ao INESC TEC, pelo apoio deste doutoramento através do seu financiamento pluri-anual durante 3 anos. À Fundação para a Ciência e a Tecnologia (FCT), por ter apoiado o último ano através da bolsa de doutoramento (SRFH/BD/114842/2016). Por fim ao Departamento de Informática da Universidade do Minho e ao HASLab pelas condições necessárias ao desenvolvimento deste trabalho.

Braga, Janeiro de 2018
Fábio Coelho

The important thing is to not stop questioning.

Curiosity has its own reason for existing.

Albert Einstein

Towards a transactional and analytical data management system for Big Data

Hybrid database systems are on the verge of making *Big Data* analytics a reality. This new class of database systems bypasses traditional methodologies considered to update data on the analytical processing engine, moving such processing to be computed directly on top of production data. Uncovering a unified database engine that can achieve scalable analytics while simultaneously keep a steady operational capacity, needs to overcome some of the current system hurdles, namely the Extract, Transform and Load (ETL) process. By eschewing such process, hybrid database engines are poised to reduce implementation, management and storage costs and ultimately, enabling *real-time Big Data* analytics.

This dissertation addresses hybrid database systems, particularly tackling some of the inherent functional and non-functional challenges associated with the provision of *real-time* analytics. This was achieved by specializing in a particular class of analytical functions designated as *Window Functions*. We considered this class of analytical functions as a vehicle to understand and address the low-latency requirements in hybrid systems, by considering a highly scalable and cloud-based operational database as foundation. While we equipped it with the ability to compute analytical functions, new algorithms were developed to account for the highly distributed scenario. We devised a new metric and evaluation system specifically targeted to assess hybrid database systems, showing that the accomplished prototype is able to meet current requirements. Each one of these achievements is presented as a novel contribution that addresses the proposed challenges and unravels the path for a *real-time* analytics database.

Rumo a sistemas de gestão de dados transacionais e analíticos para Big Data

As bases de dados híbridas estão prestes a tornar o processamento de dados analíticos em *Big Data* numa realidade. Esta nova classe de bases de dados evita as metodologias tipicamente consideradas para a atualização de dados nos motores de processamento analítico, movendo-o para ser computado diretamente sob a base de dados operacional. Alcançar uma base de dados híbrida, munida de um motor unificado que possibilite processamento analítico escalável e seja simultaneamente capaz de manter um nível de processamento operacional estável, terá necessariamente que ultrapassar alguns dos obstáculos hoje encontrados, nomeadamente o processo de transformação de dados, do Inglês (ETL). Ao evitar este processo, as bases de dados híbridas terão um papel ativo, reduzindo custos de implementação, gestão e armazenamento, o que em última análise promoverá o processamento analítico de *Big Data* em *tempo-real*.

Esta dissertação centra-se em bases de dados híbridas. Em particular, aborda alguns dos desafios funcionais e não-funcionais associados ao aprovisionamento de uma capacidade de processamento analítico em *tempo-real*. Nomeadamente, recorreu-se a uma classe de funções analíticas designadas por *Window Functions* (Funções em Janela), considerado-as como veículo à percepção e adoção de requisitos inerentes ao processamento híbrido. Considerou-se desta forma uma base de dados operacional altamente escalável, fundada em tecnologias orientadas ao processamento na nuvem. Partiu-se para a adequação do referido motor de base de dados, por forma a equipa-lo com a capacidade de interpretação e execução desta classe específica de funções analíticas, enquanto novos algoritmos foram desenhados por forma a considerar o ambiente altamente distribuído em que esta base de dados se insere. Desenvolveu-se uma nova métrica e plataforma de avaliação, inovadora na forma como as distintas distribuições de pedidos (transacionais e analíticas) são combinadas numa única, capaz de avaliar sistemas de bases de dados híbridas. Esta métrica serviu posteriormente para demonstrar que o pro-

tótipo desenvolvido está capacitado para responder aos desafios propostos. A realização de cada um dos objétivos propostos está apresentado como uma nova contribuição científica, contribuindo para desbravar o caminho com vista a um sistema integrado e capacitado para o processamento em *tempo-real* de *Big Data*.

Contents

Acronyms

ACID Atomicity, Consistency, Isolation and Durability. 2, 3

BI Business Analytics. 20, 21

CEP Complex Event Processing. 79

DOP Degree Of Parallelism. 63

DQE Distributed Query Engine. 62, 63, 69, 70, 73–76

ETL Extract, Transform and Load. 3, 4, 15, 21, 49, 109

HDQE Hybrid Distributed Query Engine. 5–7, 9, 81–83, 102, 105

HTAP Hybrid Transactional and Analytical Processing. 4, 6, 11, 20, 21, 24, 27, 29, 32, 36, 38, 41, 44, 45, 52, 81, 109, 110

I/O Input/Output. 3, 13, 16, 49

IoT Internet of Things. 81

JDBC Java Database Connectivity. 36, 82, 88

MR Map-Reduce. 3, 15

MVCC Multi Version Concurrency Control. 21

OLAP OnLine Analytical Processing. 2, 4, 11, 14–16, 20, 21, 23, 26, 32, 33, 36–42, 44, 52, 81, 102, 103, 109, 110

OLTP OnLine Transaction Processing. 2–4, 11, 14–16, 20, 21, 26, 30, 32, 33, 35, 37–42, 44, 52, 81, 102, 109

PaaS Platform as a Service. 81

PBK Partition By Key. 74, 75

QE Query Engine. 62, 63, 110

QO Query Optimizer. 62

RDBMS Relational Database Management Systems. 2, 12

SPC Storage Performance Council. 48

SPEC Standard Performance Evaluation Council. 48

SQL Structured Query Language. 3, 5, 6, 9, 15

SUT System Under Test. 24, 26, 28, 36–38, 104, 105

TPC Transaction Performance Council. 48

TPC-C Transaction Processing Performance Council Benchmark C. 23, 30, 31, 33, 35, 36, 38, 48, 72

TPC-DS Transaction Processing Performance Council Benchmark DS. 23, 49, 73

TPC-E Transaction Processing Performance Council Benchmark E. 23, 49

TPC-H Transaction Processing Performance Council Benchmark H. 23, 29, 32, 34, 35, 49, 103

WF Window Functions. xix, xx, 9, 55, 66, 78–81, 103, 104, 106, 110

List of Figures

List of Tables

Chapter 1

Introduction

With the establishment of *cloud computing* as a mainstream technology, companies worldwide acquired a significant grasp on how this technology could improve their day-to-day activities. The ubiquity brought by this new paradigm made access to all kinds of data, be it documents, presentations, photographs, videos or other raw formats widely accessible due to the immediate and continual availability provided. The storage, structuring and indexing of these huge amounts of data for subsequent efficient access and processing have been a major business opportunity, but also a great challenge for cloud service providers. At the same time, the high availability standards and competitive offerings for cloud storage services and processing capabilities allure an increasing number of businesses to migrate their systems into the cloud, reducing maintenance and ownership costs. This adds up not only to the sheer volume of data to be managed by cloud providers, but also, to the diversity of solutions that are currently offered.

In a parallel path, data analytics experimented a *Big* progress, boosted by the scalability offerings of the cloud computing trend, from where the contemporary term *Big Data Analytics* was coined. Past years saw a meaningful growth of related technologies as some of the largest Internet players such as Google, Facebook, Amazon or Microsoft invested large sums of capital to expedite such technologies. In a recent study, IDC, the International Data Corporation, estimates that by the end of 2020, the investments covering Big Data Analytics and associated technologies could reach 200 billion USD (United States Dollar) [IDC, 2015]. Despite the huge data volumes already handled by such companies, the techniques considered are often extremely inefficient and coupled together in an ad-hoc manner, consuming tremendous amount of resources which in turn results in a very high total cost of ownership [Stonebraker, 2008].

Nowadays, the heterogeneity found in the universe of *Big Data* applications is the consequence from tailoring applications to suit a subset of *Big Data* problems, such as the increase in data volume for data management systems or the processing rhythm for real-time systems. Regarding data management, it is possible to observe the flourishing of new data technologies that can somehow be better suited to fulfill the needs of the *Big Data* scenario (*e.g.,* distributed databases within the Hadoop [Shvachko et al., 2010] ecosystem), a significant part of all data is still being stored in data repository systems such as Relational Database Management Systems (RDBMS).

Traditionally, RDBMS are associated with a set of guarantees that surround operations in a safety net in what concerns to data consistency and durability. However, this safety net is also associated with failing to comply with some of the key non-functional characteristics of the cloud computing paradigm: the high availability and scalability. RDBMS have now moved from centralized and monolithic architectures to profit from the elasticity provided by cloud-based platforms, while maintaining focus on data consistency.

RDBMS address the maintenance of some key characteristics: Atomicity, Consistency, Isolation and Durability (ACID). It focuses on the transaction abstraction, enabling OnLine Transaction Processing (OLTP) systems to keep concurrent operations (*e.g.,* inserts and updates) consistent. Nonetheless, the increasing trend to perform *Big Data* analytics as part of a decision support system, places a burden over OLTP systems as they are one of the main data sources for decision support systems and, the requirements for maintaining data consistency conflict with the ideal requirements for analytical workloads, such as handling multi-dimensional queries.

OnLine Analytical Processing (OLAP) systems are typically considered in parallel, as a way to overcome the shortcomings presented by OLTP systems regarding data analysis. Such systems are fed with data from a variety of sources, namely from OLTP systems, and converted to a multi-dimensional model. This data model fits the type of analytical queries as stored data is already aggregated according to a schema.

To achieve this multi-dimensional aggregated schema, data undergoes a transformation procedure commonly referred to as Extract, Transform and Load (ETL). The process considers sometimes a set of production databases to perform the extraction of data, then performing the required transformations (*i.e.,* combining several entities into a large dimension) and finally loading it in a multi-dimensional database that will be ready to accept analytical queries. The transformation that data undergoes during this procedure is usually very greedy in terms of completion time and computational resources. The overhead imposed by the ETL process is specially acute when data feeding this process originates from an OLTP system – typically a Structured Query Language (SQL) ACID database. This procedure can be so disruptive to the performance of the OLTP system that its actions are usually scheduled to periods of under utilization.

Real-time analytics has now reached a level of pervasiveness beyond expectation, reaching society in new fields like social media analysis, online advertising or more traditional domains such as trading and stock markets [Ellis, 2014, Bange et al., 2013]. This is actually perceived as being able to collect insights and trigger responses *as fast as* data is made durable [Liu et al., 2014]. The high availability provided by the cloud computing paradigm created a good nest for several distributed computing technologies, particularly the Hadoop/Map-Reduce (MR) framework [Shvachko et al., 2010, Dean and Ghemawat, 2008], that observed a large adoption and fork rate during the last decade. However, this framework was designed with scalability and fault-tolerance in mind, being fine-tuned for throughput and not for Input/Output (I/O) efficiency [Lee et al., 2012], which does not meet the need for low-latency operations. Moreover, the high response times brought by the ETL procedure reduces data analysis to a series of snapshots of data. Together, these concerns impose limits to the key aspects of a true *real-time* processing system.

Database systems are undergoing a design change, boosted by the need for achieving real-time analytics over production data. Hybrid database systems introduce an opportunity to merge the transactional and analytical workloads. The lookout for a hybrid solution that is simultaneously able to fulfill operational and analytical workloads, must comply with low-latency operations, leveraging all parallel opportunities and ultimately eschewing the ETL.

1.1 Problem Statement and Objectives

The increasing demand for real-time analytics requires the fusion of Transactional (OLTP) and Analytical (OLAP) systems and has been sparking several proposals for the so-called Hybrid Transactional and Analytical Processing (HTAP) systems. As pointed out by Gartner [Pezzini et al., 2014], this new class of database engines should be capable of handling mixed workloads with high levels of transactional activity and, at the same time, providing scalable business analytics directly over production data. By eschewing the ETL process, HTAP systems are poised to reduce implementation, management and storage costs and, most importantly, enable real-time analytics over production data.

This thesis is precisely focused in bridging the gap between OLTP and OLAP systems, addressing some of the current shortcomings found in today's attempts to unlock HTAP database engines. For instance, it is not trivial to assess hybrid database systems. As OLTP and OLAP database systems have been kept separated, the respective evaluation systems have also been kept separated, focusing in the assessment of the particular aspects of each workload type. Likewise, the first objective is to pursue the understanding of how to assess a hybrid database engine regarding its functional and non-functional properties, in order to validate architectural decisions.

Hybrid database systems introduce considerable new challenges when merging OLTP with OLAP workloads, particularly the ingestion of newly produced data from the transactional activity into the analytical execution, without ETL. Therefore, the second objective is to understand how data aggregations can reflect newly ingested items from the transactional activity by addressing a particular class of analytical functions.

HTAP systems suggest new architectural challenges associated with the provision of low latency analytics. Likewise, the third objective is to leverage the research in parallel-distributed architectures, enabling analytical SQL queries in parallel over a distributed *shared-nothing* system.

1.2 Contributions

This dissertation introduces three novel contributions. The first contribution arises from the need for a thorough evaluation of hybrid database engines. Current benchmarking approaches are not able to comprehensively produce a metric – described ahead as *unified metric* – that can convey all the functional characteristics of a hybrid workload. The evaluation of operational and analytical database engines was so far achieved through disjoint workloads. A new benchmark suite, HTAPBench is proposed, providing a new unified metric for hybrid systems.

As a second contribution, we particularly address a specific class of analytical functions designated as *Window Functions*. This subclass of analytical operators allow data to be handled in a derived view of a given relation, considering an execution window, built from a configurable array of consecutive tuples. This class of analytical functions is keen to create logical data partitions defined in a per-query basis that needs to be mapped to the actual physical partitions of data. The efficiency of these functions is bound to the ability to establish a high affinity among the elements of each logical partition. This contribution introduces a holistic mechanism that learns how to exercise the affinity between elements of logical partitions by looking at the physical location of each logical partition.

The third contribution leverages the research in parallel-distributed databases, enabling the execution of parallel analytical SQL queries over a distributed *shared-nothing* system. The understanding that logical partitions may be arranged and forwarded in special batches improved the previous considered affinity mechanism. This is achieved by exploring a correlation strategy that analyzes the Similarity between elements of a given logical partition. The effectiveness of the mechanism was then verified during the distributed execution of queries, showing how data similarity can be employed across partitions to improve data co-locality between nodes of a distributed database. The mechanisms are integrated in a SQL Hybrid Distributed Query Engine (HDQE), equipping it with the ability to compute this class of analytical functions in a setup composed from a configurable number of instances. A thorough evaluation is provided, considering a series of micro-benchmarks and considering the proposed benchmark suite.

1.3 Software Prototypes

We developed prototypes for all the previously described contributions:

- We have implemented HTAPBench, a benchmarking suite particularly designed to assess HTAP database systems. The evaluation and validation of results show that it is able to provide a unified metric to evaluate a common transactional and analytical workload.

Available at `https://github.com/faclc4/HTAPBench`

- We started from the Apache Derby query engine and fully implemented its capability to interpret and execute SQL *Window Functions*. The prototype modified the SQL parser and cost optimizer, tailoring the solution to be sensitive to the statistical mechanisms considered.
- We implemented prototypes for the Holistic and Similarity statistical mechanisms, considering simulated configurations that emulated a distributed query engine.
- We have implemented the Holistic and Similarity mechanism in the HDQE as part of its underlying data store. For this purpose we particularly leverage the CoProcessor Framework found in HBase.

1.4 Publications

The work presented in this thesis has been published in a number of publications in distinct international conferences:

- Fábio Coelho, João Paulo, Ricardo Vilaça, José Pereira, Rui Oliveira. HTAPBench: Hybrid Transactional and Analytical Processing Benchmark. In *Proceedings of the 8th ACM/SPEC International Conference on Performance Engineering (ICPE)*, 2017

  This conference paper presents the architecture of HTAPBench, a new benchmarking suite designed to bridge the gap of current benchmarking solutions, geared towards the assessment of hybrid database engines. HTAPBench introduces a new client balancer that controls transactional and analytical client flows and assembles the results in a clever new unified metric.

- Fábio Coelho, José Pereira, Ricardo Vilaça, Rui Oliveira. Holistic Shuffler for the Parallel Processing of SQL Window Functions. In *Proceedings of*

16th IFIP Conference on Distributed Applications and Interoperable Systems (DAIS), 2016

This conference paper analyzes and expedites statistical properties observed in a distributed key-value store when computing a class of analytical functions designated as Window Functions. The translation of these observations into a histogram, enables a key-value store that is configured as the data substrate of a distributed SQL query engine to judiciously forward data to the nodes where the computation was assigned.

- Fábio Coelho, Miguel Matos, José Pereira, Rui Oliveira. Similarity Aware Shuffling for the Distributed Execution of SQL Window Functions. Best Paper Award in *Proceedings of 17th IFIP Conference on Distributed Applications and Interoperable Systems (DAIS)*, 2017

  This conference paper builds on previous work and pushes the query engine technology forward, by understanding similarities between remote data partitions. The added knowledge enables the query engine's data forwarding mechanism to efficiently manage the bandwidth usage when transferring data between remote nodes during distributed execution.

The following work is ready to be submited:

- Fábio Coelho, Rui Gonçalves, Miguel Matos, Francisco Cruz, José Pereira, Rui Oliveira. Parallel SQL Window Functions in Distributed Databases.

  This conference paper draft introduces the previously described new strategies in a HDQE, particularly tackling all parallelization opportunities. This was achieved by enabling several workers to split and share query execution, while choosing the optimum execution location that minimizes data forwarding among workers.

Also, preliminary work has been published that greatly improved our knowledge of transactional processing mechanisms on top of NoSQL key-value stores:

- Fábio Coelho, Francisco Cruz, Ricardo Vilaça, José Pereira and Rui Oliveira. pH1: A Transactional Middleware for NoSQL. In *Proceedings of 33th IEEE International Symposium on Reliable Distributed Systems (SRDS)*, 2014

This conference paper introduces a new framework, *pH1*, that brings the transactional wrapping agent atop NoSQL Key-Value stores. The framework was assessed with two NoSQL stores, Cassandra and HyperDex, showing an overall limited performance impact. Moreover, the extensible characteristics of the framework allow it to be easily plugged-in with other NoSQL stores.

Additionally, the result of collaborations paving the way for this thesis or leveraging its research appear in the following publications:

- Pascal Felber, Marcelo Pasin, Etienne Rivière, Valerio Schiavoni, Pierre Sutra, Fábio Coelho, Miguel Matos, Rui Oliveira and Ricardo Vilaça. On the Support of Versioning in Distributed Key-Value Stores. In *Proceedings of 33th IEEE International Symposium on Reliable Distributed Systems (SRDS)*, 2014

  This conference paper presents a study of the design and implementation space for providing versioning support on top of a distributed key-value store. A versioning API that enables multiple writers is provided as common Key-Value stores do not offer the necessary synchronization power.

- Francisco Maia, João Paulo, Fábio Coelho, Francisco Neves, José Pereira and Rui Oliveira. DDFlasks: Deduplicated Very Large Scale Data Store. In *Proceedings of Distributed Applications and Interoperable Systems (DAIS)*, 2017

  This conference paper extends DataFlasks, an epidemic data store for massive scale systems, introducing deduplication mechanisms. The design and evaluation are provided and the results are twofold. First deduplication is able to decrease storage requirements and also decrease network bandwidth, while maintaining a fully-decentralized and resilient design.

1.5 Outline

The rest of the document is structured as follows:

Chapter 2 starts by describing the meaningful characteristics of operational, analytical and hybrid databases.

Chapter 3 introduces HTAPBench, a new benchmarking suite designed to evaluate hybrid database systems, particularly addressing its design, workload and component specification. Afterwards, this new benchmarking suite is evaluated and the results validated. This new suite offers a new unified metric that is able to slot the results in a quadrant field plot.

Chapter 4 introduces WF as the main class of analytical functions covered in this dissertation, considering two new techniques that are able to expedite the distributed execution of this class of analytical functions.

Chapter 5 presents the architecture for the SQL HDQE that is simultaneously able to sustain an operational workload at a steady throughput, while at the same time perform analytical WF.

Chapter 6 concludes this dissertation, discusses its achievements and addresses future research paths.

Chapter 2

Background

This chapter covers several meaningful aspects regarding the classification of database systems. First, we consider key concepts behind the current classification of database systems. We address OLTP systems, covering transactional semantics and considered data structuring. Moreover, we address OLAP systems, also covering data structuring and the main steps taken toward the scalability of their operations. Afterwards, we discuss the main features and challenges in current HTAP approches, discussing what should be taken into account when merging both workloads types into a hybrid one.

2.1 Online Transactional Processing

OLTP systems set a type of data processing that is the standard in today's enterprises. As the name suggests, such systems are based in the transactional perspective of data management. An OLTP system is characterized by having a large amount of rather small transactions, namely: insert, update or delete operations. The focus of such systems is to achieve a fast query processing mechanism, while preserving data integrity by enclosing operations in a transaction.

A transaction is a sequence of operations whose execution traditionally satisfies the following ACID [Haerder and Reuter, 1983] properties:

- **Atomicity**: Either all or none of the operations within a transaction are successfully performed.

- **Consistency**: Transactions preserve system constraints, as a whole. Any transaction takes the system from a valid state to another valid state.

- **Isolation**: The concurrent execution of transactions preserves the semantics of the defined correctness criterion or isolation level.

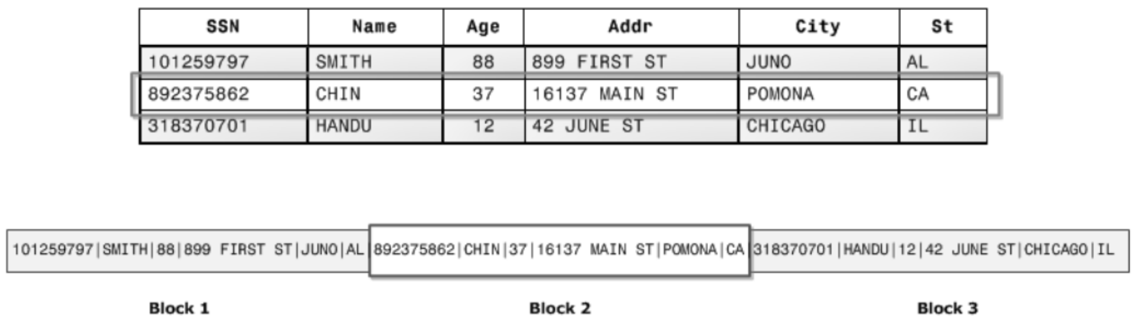
- **Durability**: The effects of a successful transaction are durable even in the presence of faults.

A transaction that performs successfully is said to *commit*, otherwise it *aborts.* A transactional system enables the coexistence of concurrent transactions which may lead to several incidents related to data being accessed and modified concurrently. These events called anomalies are described in [Berenson et al., 1995] and they are used to characterize four isolation levels, formally described in the ANSI SQL standard. Namely, these isolation levels are (in ascending order from the least to the most strict level): *read uncommitted*, *read committed*, *repeatable read* and *serializable.*

With the existence of concurrent transactions, transactional systems rely on concurrency control mechanisms to avoid anomalies. These mechanisms can be based on mutual exclusion primitives, establishing *read locks* over data that will be read and, *write locks* over data that will be written. Thus, if a transaction acquired, for instance, a write lock over a data item, for the period of time that the lock is held, no other concurrent transaction will be able to acquire a write lock over the same piece of data. The different isolation levels define distinct possible behaviors regarding the acquisition of read or write locks over a data item by concurrent transactions, in order to avoid the described anomalies.

The most strict isolation level – *Serializable* – does not allow for any type of anomalies to occur [Berenson et al., 1995]. With a concurrency control mechanism based on mutual exclusion, locks ensure the absence of anomalies, as in the *Serializable* isolation level. To provide this isolation level, the *Two-Phase-Lock* protocol [Bernstein et al., 1986] is usually used to control the acquisition and lock release.

Currently the default isolation level in most commercial RDBMS is called *Snapshot Isolation* [Berenson et al., 1995, Ports and Grittner, 2012]. It uses both multi-version concurrency control and timestamps in order to avoid locks [Revilak et al., 2011], allowing a transaction to work over a consistent snapshot of data. This is presented as one of the main advantages of this isolation level, as



SSN	Name	Age	Addr	City	St
101259797	SMITH	88	899 FIRST ST	JUNO	AL
892375862	CHIN	37	16137 MAIN ST	POMONA	CA
318370701	HANDU	12	42 JUNE ST	CHICAGO	IL

101259797|SMITH|88|899 FIRST ST|JUNO|AL
892375862|CHIN|37|16137 MAIN ST|POMONA|CA
318370701|HANDU|12|42 JUNE ST|CHICAGO|IL
Block 1
Block 2
Block 3

Figure 2.1: Row-oriented data layout.

a transaction is never blocked performing a read operation (which is the case for concurrency control based on mutual exclusion), potentially increasing the level of concurrency. Snapshot Isolation is usually the most strict isolation level found in RDBMS, and for a wide array of applications it is possible to achieve a serial execution [Fekete et al., 2005] or even to fully implement serializability [Cahill et al., 2009].

Data placement strategy

Database systems organize their data into tabular structures composed of several rows built from a set of column qualifiers. Typically, columns are used as containers to characterize a given feature, while rows are used to report occurrences.

When a transaction modifies one or a group of columns regarding a given entry (*i.e.,* a row), the database system needs to coordinate with the underlying operating system how and when the I/O device should be accessed. The I/O device physically stores data in contiguous equally sized blocks. Moreover, the layout adopted by the I/O device plays an important role that is directly related with the workload characteristics.

To reduce the frequency and the number of blocks the I/O device needs to access, database systems are usually faced with the choice to either use *row-oriented* or *column-oriented* data layouts. Figure 2.1 depicts a*row-oriented* data layout. Therefore, the database system adopts a data layout where the records that account for each row are stored sequentially, using as many blocks as needed to hold the data. Such trend provides advantages for transactional database systems as a given transaction is likely to access and modify several attributes of a single row at each cycle. Thus, a *row-oriented* storage strategy reduces the number of cycles required to acquire the required blocks, better suiting the needs for OLTP systems.

2.2 Online Analytical Processing

OLAP sets a contrasting approach to transactional systems, allowing to answer multi-dimensional queries. This class of systems is usually considered for reporting and analysis of data spread across several individual dimensions. Actually, the border line from data warehousing methodologies and OLAP systems is not sharp, as they are intrinsically related. Data warehousing approaches can be seen as the data layer that will serve as base to analytical tools. Most of the data requirements found in them end up falling into the specific details of multi-dimensional data modeling.

A multi-dimensional schema hold tables as dimensions which relate among them through structures called fact tables, which in turn agglomerate several measures. OLAP systems use this structure to build the OLAP cube, a multi-dimensional representation that enables to visualize how dimensions are correlated. Each intersection in the cube represents a single measure which extends through the dimensions as a vector space.

OLAP systems are usually categorized according to the following taxonomy:

- **ROLAP**: These systems allow the execution of analytical queries over relational storage. The relational database in such systems, besides holding tables for the desired dimensions also adds further tables that enable querying aggregated data. The main advantage of ROLAP systems is that they do not limit the possible business questions, as no restrictions on the dimensions of the OLAP cube are imposed. However, as queries are answered straight out of the operational database, the transactional operation may suffer delays and throughput penalties.

- **MOLAP**: This definition stands for Multi-dimensional OLAP and it is the classic configuration of OLAP systems. This approach considers a multi-dimensional data model, rather than considering a relational model. The latter requires the OLAP cube to be pre-computed. MOLAP systems present great performance for query execution, which is due to the indexing and caching capabilities found in multi-dimensional systems. However, MOLAP has major negative points due to the need for data processing through the use of the ETL process and difficulties related with the ability to efficiently handle dimensions with high cardinality.

Scalable Analytics

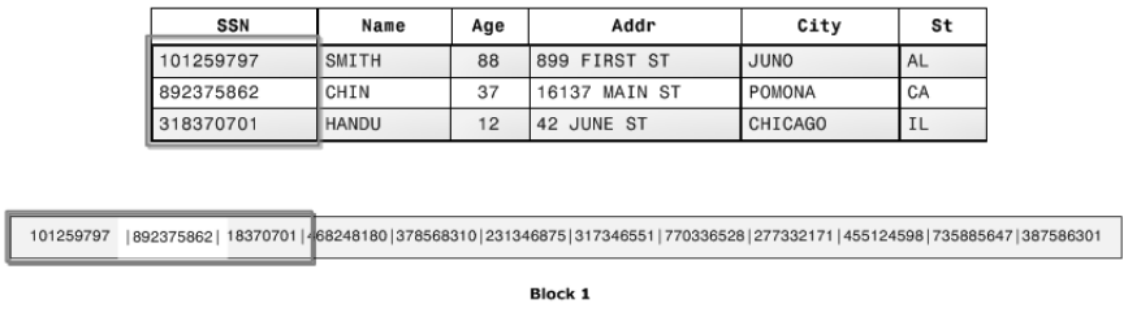
There are currently two main trends to provide scalable analytics. The first trend follows the MR approach [Dean and Ghemawat, 2008] with improvements to bring it closer to SQL semantics. The standard SQL MR introduced by Hadoop [Shvachko et al., 2010] is a programming model based in two functions: the *map* and *reduce*. With the *map* function, users generate an intermediate set of key/value pairs from the data set, while the *reduce* function merges all the intermediate values associated with each key. MR is targeted to *Big Data* sets and currently offers several advantages over the use of parallel database systems in what regards storage system independence and fine grain fault tolerance. BigQuery [Tigani and Naidu, 2014], Tenzing [Chattopadhyay et al., 2011] or Hive [Thusoo et al., 2010] provide a SQL interface over a MR framework and a scalable key-value store. BigQuery allows only a subset of SQL operators with basic data aggregation and projections. Hive maps operators like equi-joins or unions to MR jobs. Tenzing also relies on MR to provide a query language closer to SQL. Such projects are OLAP oriented and as a consequence, they require ETL procedures, generating duplicated data and harming *real-time* analytics.

The second trend leverages parallel-distributed databases that fully support SQL semantics, making it more expressive when compared with the former MR approach for normalized datasets. Nevertheless, solutions for this approach such as Vertica [Lamb et al., 2012] or GreenPlum [Waas, 2008] are strict OLAP oriented projects, requiring ETL and lacking the ability to perform OLTP workloads also relying on the ETL approach.

Data placement strategy

OLAP systems provide a query centric approach to compute data aggregations. Unlike OLTP systems, where transactions span over a sub-set of the available column qualifiers, the queries in OLAP systems are commonly restricted to a single or a small set of column qualifiers.

Figure 2.2 depicts a *column-oriented* data layout, where each distinct element is serialized in batches fed from sequences composed by elements in each column qualifier.



SSN	Name	Age	Addr	City	St
101259797	SMITH	88	899 FIRST ST	JUNO	AL
892375862	CHIN	37	16137 MAIN ST	POMONA	CA
318370701	HANDU	12	42 JUNE ST	CHICAGO	IL

101259797 |892375862| 18370701 |468248180|378568310|231346875|317346551|770336528|277332171|455124598|735885647|387586301

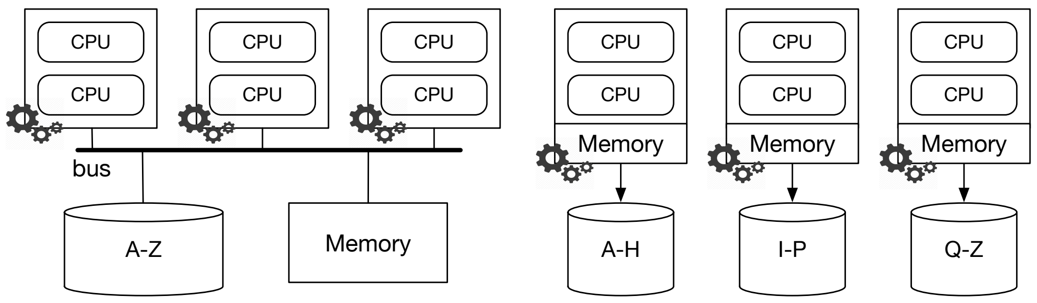
Figure 2.2: Column-oriented data layout.

A single aggregation (*e.g., sum(x)*) considers all the occurrences (*i.e.,* rows) of a given attribute (*i.e.,* column qualifier) and computes the aggregated value. The *column-oriented* data layout allows to sequentially read the data elements regarding a single aggregation with the least required effort, as the data layout matches the read pattern. If analytical systems consider the same *row-oriented* strategy as the transactional system, the operation would generate a significant larger number of calls to the underlying I/O system. That is, *column-oriented* data layouts keep each column individually, enabling queries to read only the required column and not having to acquire full data rows and afterwards discard the unneeded attributes. Moreover, OLAP systems benefit from *column-oriented* layouts as it allows for better data compression [Abadi et al., 2013], since typically, sequential elements show smaller deltas between them when compared to *row-oriented* layouts.

2.3 Discussion

OLTP and OLAP systems establish distinct goals that are intrinsically related with their target workloads. Beyond the limits imposed to the ideal data layouts, the underlying architecture also impacts on the scalability of the system, thus restricting approaches for parallel processing.

The parallel execution of tasks in database systems can be classified in two different classes in respect to how data is stored and acquired for processing. The two strategies are often referred in the literature as: *shared-everything* and *shared-nothing* systems. The key difference between these two approaches lies in how physical data is partitioned and accessed for processing. Figures 2.3(a) and 2.3(b) depicts both categories.



(a) Shared-everything. (b) Shared-nothing.

Figure 2.3: Shared architectures for distributed computing.

In a shared-everything system, no data partitioning strategy is considered. Data is stored as a whole in a single storage system. Hence, the computing nodes involved in completing a given database task will share the same data source. By opposition, in a shared-nothing system, data is physically split among several devices. In principle, and considering no replication, a range of the data spectrum only exists in a single location. Each computing node in such a system would then only have access to its partition.

Consider, as an example, a database that would store all the first names of citizens in a country. The shared-everything system has all the names started from 'A' to 'Z' in the same data location. The shared-nothing system partitions the key range across several locations. Considering three distinct nodes, each one would roughly handle a third of total amount of keys. To achieve such behavior, the partition of data has to be planned upfront, usually by using a hashing algorithm. The hashing algorithm will take into consideration the range of keys and the number of partitions in order to evenly split the key range by each data location. As the computing nodes in a shared-nothing architecture only have access to a particular partition location, they necessarily have to be coordinated in order to only receive requests concerning the range of keys they own.

Hashing algorithms are usually considered to divide the key range according to the required amount of data partitions. They are also required to provide a coordinator with information regarding where each data tuple is stored. Fine tuning of the hashing algorithm becomes decisive to the overall performance of the system. The adequate fragmentation of data and allocation of the database is then key to whole system. As each data node can only access its own partition of data, poorly deciding on how data is partitioned could have severe consequences, as it largely determines where the database operations are performed, thus affecting node utilization and communication. As complex queries could possibly target several data partitions and multiple possible data distributions, the partitioning strategy must inevitably constitute a compromise for a specific workload.

Trade-offs for write operations

In a shared-everything architecture, since all computing nodes share the same data source, write operations can be performed by any node (*i.e.,* no coordination is required to redirect operations to specific nodes). This may possibility cause a consistency problem if two or more nodes attempt to update the same data item. To prevent this, the management system requires lock mechanisms or to communicate its intentions with all the participant nodes, which may impair the scalability of the system.

Shared-nothing architectures, in principle, do not suffer from any issues related with distributed locks, since a given tuple can only be modified by one computing node (provided that the request is correctly forwarded to the correct node). From the perspective of write operations, a shared-nothing architecture is better positioned to scale linearly. To ensure data consistency, transactions that need to access several partitions require a distributed coordination primitive, ensuring that a write operation spanning more than one partition is done atomically.

Trade-offs for read operations

The constraints imposed to both architectures when performing read operations are different from the ones just presented. First, as shared-everything architectures consider the same data source for all requesting nodes, some level of contention may be expected as the number of nodes accessing data grows. Second, the read throughput may be affected in cases where cache sizes are not correctly adjusted. This opposes to the shared-nothing architecture as in principle, a given node would only have to cache its partition (roughly $1/n$ being n the number of nodes). Shared-nothing architectures present one major hurdle related with queries having to read data from other partitions besides their own. This is particularly true considering aggregations that need to contact all data partitions, which clearly becomes prohibitive for data systems that should be able to scale.

Therefore, while performing read operations, shared-everything architectures could face performance degradation due to contention and inefficient use of caches. Shared-nothing architectures could potentially suffer from decreased scalability, if the workload imposes read operations that end up transversing all or a majority of data partitions.

From the presented trade-offs, shared-nothing architectures show several advantages to data management due to their inherent decentralized nature and scalability possibilities. Actually, there has been a current trend to introduce databases that employ the above presented considerations in their architecture. These are called NoSQL databases and they usually present a decentralized architecture and data partitioning strategies, even though they rely in weaker consistency criteria and restricted querying capabilities.

Shared-everything systems contrast with the previous as they rely in an architecture composed by a centralized data source. This architecture presents advantages in what regards to the coordination of tasks, but can place a burden to the underlying data system regarding scalability and access control.

Ultimately, the contrasts between these two types of architectures favors shared-nothing systems. These systems do not introduce resource contention as each node operates only on its data partition and has access to its own memory and operating system. The independence between nodes of a shared-nothing system also places them ahead of shared-everything architectures, showing better parallel capabilities as the system is able to scale horizontally (*i.e.,* the number of symmetric nodes grows). This is derived from a possibly larger number of tasks, each one operating a smaller and more confined dataset in each node.

2.4 Hybrid Transactional, Analytical Processing

HTAP defines a system with the capability to efficiently handle simultaneous transactional and analytical workloads. As pointed out by Gartner [Pezzini et al., 2014], this new class of database systems must be capable of delivering high levels of transactional operation (OLTP), providing at the same time scalable business analytics (OLAP) directly over production data.

Traditionally, both workloads are handled through separate engines, periodically feeding the OLAP with data from the OLTP engine through an ETL process. The approach seeks to ensure the best performance of each individual engine at the expense of data freshness for analytics. However, the ongoing trend for *real-time* analytics powering data-driven decision making is not compatible with the traditional approach to post-process analytical data offline. Instead, it requires repeatable reads over recently updated transactional data to be merged with historical data, which restricts the underlying data architectures selected for each engine.

The plethora of proposals for hybrid database systems considers mainly two approaches regarding data placement [Özcan et al., 2017], namely: employing a single system for both workload types or considering distinct systems for OLTP and OLAP.

Architectures built to perform transactional and analytical activity over a unified engine have been proposed since the dawn of the Business Analytics (BI) era. They were built from introducing the parsing capability of analytical queries into pre-existing transactional database systems (*e.g.,* IBM's DB2 [IBM, 2013] or SAP Hana [SAP, 2014]) or, the other way around, introducing optional transactional semantics into pre-existing analytical database systems. Recent updates to this architecture through database systems like Hyper [Kemper and Neumann, 2011] or Pelaton [Pavlo et al., 2017], provide the ability to re-organize the underlying data layouts from row to column-wise, or vice-versa, powering high transactional or analytical throughput according to demand. However, the use of these techniques prevents the use of the last committed data due to the data transformations required.

Architectures built from distinct database systems for each workload type, typically imply disjoint storage systems. It is then up to the database engine to establish and maintain the hybrid architecture and querying semantics. With distinct data stores, the operational data is kept in the OLTP storage (*e.g.,* row-oriented) and then fed into the OLAP storage (*e.g.,* column-oriented) through the ETL process. This has been the standard approach since the inception of BI. Likewise, the array of technologies targeting the *Big Data* environment also considers a similar layout, decoupling the storage back-ends for each system type. As NoSQL databases become a commodity technology, hybrid approaches consider these databases such as Cassandra [Lakshman and Malik, 2010] for transactional storage and then feed groomed data (*i.e.,* performing a prior data selection and aggregation) into a set of column-oriented storage back-ends for *SQL-on-Hadoop* systems, such as Parquet [Hadoop, 2017b] or ORC [Hadoop, 2017a]. Disjoint database engines may also be configured to share a common dataset. This approach has been pursued in configurations like SAP Vora [SAP, 2017], where the transactional activity is run and persisted by HANA [SAP, 2014] and the analytical activity is executed through Spark SQL [Armbrust et al., 2015] over Vora's data. Others are purely built from *SQL-on-Hadoop* systems considering HBase [George, 2011] as the transactional manager and Impala [Kornacker et al., 2015] for the analytical processing, accessing a common dataset through HDFS [Borthakur et al., 2008]. However, these systems have poor connectors to power the joint workload activity, which typically renders average transactional throughput with subpar analytical performance.

HTAP systems must overcome the limitations found in OLTP or OLAP engines, which are broadly related to their inability to scale. First, HTAP systems must accommodate Multi Version Concurrency Control (MVCC) techniques (*e.g.,* Snapshot Isolation [Berenson et al., 1995]) to enable the execution of long-running queries, avoiding contention related with the acquisition and release of locks over data. Second, these systems must be highly scalable, leveraging all parallelization opportunities and data partitioning strategies.

In spite of the previously addressed trade-offs, the upcoming chapters consider a shared-nothing architecture, as it renders better scalability opportunities for a hybrid workload.

Chapter 3

Benchmarking Hybrid Data Management Systems

Benchmarking has long lived alongside database technologies, providing estimates regarding performance, cost, dependability of specific components or a system as a whole [Gray, 1992]. Most importantly, benchmarks provide a systematic understanding on what are the meaningful, often critical, features of a system and how it should be assessed, ultimately enabling the comparison between systems.

Reasoning about a system's strengths or shortcomings directly correlates with the ability of a benchmark to rigorously quantify *how much* distinct systems differ. The answer to such question becomes domain-specific and justifies the existence of several benchmark types that account for distinct functional and non-functional requirements, defining distinct metrics in terms of the meaningful logical units they try to capture.

In the database domain and given the taxonomy of OLTP and OLAP systems, the industry along with independent organizations defined benchmarking approaches specially tailored for either transactional or analytical workloads, such as the Transaction Processing Performance Council Benchmark C (TPC-C) [Council, 2010a] and the Transaction Processing Performance Council Benchmark E (TPC-E) [Council, 2015d] for OLTP workloads and the Transaction Processing Performance Council Benchmark H (TPC-H) [Council, 2010b] or the Transaction Processing Performance Council Benchmark DS (TPC-DS) [Nambiar and Poess, 2006] for OLAP workloads. Each benchmark focuses on the optimization challenges associated with each system type, defining evaluation suites with very different and contradicting goals. This is so as optimizing an OLTP targeted operation would intrinsically degrade OLAP performance and vice-versa [French, 1995]. OLTP and OLAP workloads generate specific sets of queries that require distinct storage layouts in order to be efficient. They can also accommodate different-sized datasets, employing the concept of warehouse as scaling factor. Optimizing a storage layout to support both access patterns efficiently is not a trivial task, but it must be accomplished in order to have efficient hybrid systems. Storage accesses generated by OLTP workloads are mostly random while OLAP workloads are mainly sequential. Likewise, a hybrid workload will assess the ability of the System Under Test (SUT) to simultaneously schedule random and sequential access patterns to storage mediums and manage both light and intense operations regarding memory allocation and processor time. These are some of the reasons why these workloads have until now been evaluated independently.

Most importantly, it is not easy to combine and directly translate results from distinct benchmarks to the effectiveness of a system to handle a HTAP workload. Gartner states that a HTAP system should prioritize a sustained transactional throughput, delivering at the same time scalable analytical processing without disrupting the operational activity [Pezzini et al., 2014]. Consequently, even if both workloads can be run on the same engine, it is not straightforward to meaningfully and consistently reconcile the results of both workloads in a single HTAP metric. This is so as each workload is usually oblivious to the presence of the other, trying to independently reach the maximum qualified throughput in each separate workload, and therefore producing uncorrelated metrics.

This chapter presents HTAPBench, a new benchmark suite designed to evaluate hybrid systems with mixed OLTP and OLAP workloads. HTAPBench introduces a new unified metric for hybrid workloads. It provides a reading of the analytical capability as the system scales while a steady operational throughput is ensured. The hybrid workload proposed in this benchmark simultaneously exercises operational and analytical activity over the same system. The operation is governed by a new client balancer that controls how and when analytical clients are launched, ensuring that the OLTP activity stays within a configured threshold and that the results are kept comparable across runs by addressing data uniformity of the workload. The current chapter considers a specific nomenclature: *database* when referring to the stored data, *engine* when referring to the software and *SUT* or *system* when referring to the composition of software and underlying hardware.

The remainder of this chapter is organized as follows: section 3.1 introduces the design and workload of HTAPBench. Section 3.2 presents the considerations taken into account, enabling the analytical operation to consider newly modified data by the transactional part of the workload. Section 3.3 specifies all the components within HTAPBench, in particular the new unified metric proposed. Moreover, section 3.4 deploys a benchmarking campaign, aiming to demonstrate, evaluate and validate the meaningfulness of the benchmarking suite. Section 3.6 reviews relevant work for this chapter and section 3.7 offers a discussion of the achieved results.

3.1 HTAPBench Design

The Hybrid Transactional and Analytical Processing Benchmark was designed to assess database engines capable of delivering mixed workloads composed of OLTP transactions and OLAP business queries without resorting to ETL. Typically, in environments with mixed workloads, the relative weight given to OLTP and OLAP is governed by delivering a high OLTP throughput while still being able to simultaneously perform analytical business queries Pezzini et al. [2014]. This goal should be met in such a way that the OLTP throughput is kept within expected intervals. Likewise, HTAPBench focuses its operation on ensuring a stable OLTP throughput and assessing the capability of the SUT to cope with an increasing demand on the OLAP counterpart.

Transactional systems conform to a group of well defined properties often referred by ACID: Atomicity, Consistency, Isolation and Durability. These properties ensure that the integrity and consistency of data is maintained despite faults or concurrent accesses; and have been the keystone of traditional relational database management systems [Garcia-Molina, 2008]. The underlying *system* is therefore required to provide operational activity governed by these properties.

The design of HTAPBench is composed of several modules as depicted in Figure 3.1. The Density Consultant, the Client Balancer and the Dynamic Query-H Generator modules provide the foundation of this approach and are discussed in this section.

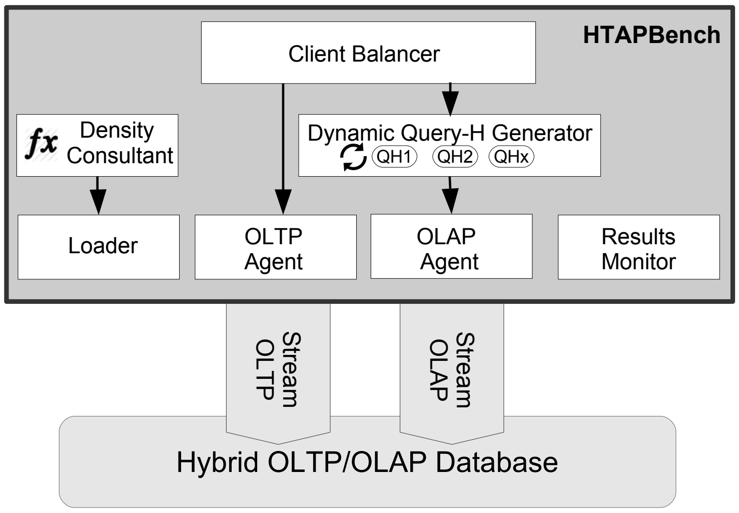


Figure 3.1: HTAPBench architecture.

HTAPBench decomposes the execution into three main stages as depicted in Figure 3.2: (*i*) the populate stage, (*ii*) the warm-up stage and (*iii*) the execution stage. Two of the modules, which are defined as agents, will regulate the OLTP and the OLAP activity. During system start, HTAPBench will be configured with a target OLTP throughput, triggering an OLTP workload configured with the required number of clients to meet the required throughput.

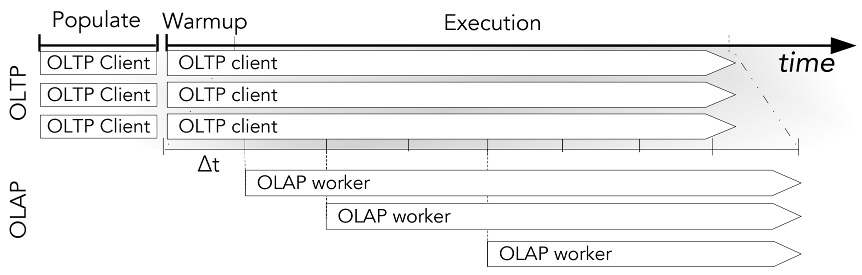


Figure 3.2: HTAPBench execution.

Periodically, HTAPBench will assess the ability of the SUT to handle an increasing OLAP activity, while ensuring that the transactional throughput does not decrease below a configured threshold.

The unified metric is central to the design, and its genesis is directly translated from the need of HTAP systems to scale without disturbing the OLTP activity. It mirrors the ability of a given analytical worker to complete queries, in a scenario composed of an increasing number of analytical workers and a stable transactional activity. Analyzing this behavior will enable the identification of situations where adding an additional OLAP worker degrades the OLTP/OLAP engine performance.

3.1.1 Workload

The mixed workload used in this benchmark is composed of a transactional agent and an analytical agent that simultaneously instruct the system to perform operations over the same dataset.

TPC-C and TPC-H were respectively selected as the operational and analytical agents, as each one is able to stress the inherent characteristics of each workload type. TPC-C was chosen due to its high rate of read-write operations, being one of the most used workloads for OLTP evaluation. TPC-C specification models a real-world scenario where a company, comprised of several warehouses and districts, processes orders placed by clients. The workload scales according to the number of configured warehouses.

TPC-H also specifies a real-world scenario, modeling a wholesale supplier and employing a schema that is very close in structure to TPC-C. Moreover, we selected TPC-H over TPC-DS [Council, 2012] since the workload in TPC-DS is data warehouse-driven, not only relying on a star schema but also requiring the use of ETL to keep data updated and in conformity with such a schema. On the basis that analytical queries in a hybrid workload should exercise a dataset common to the operational workload, TPC-H better fits the requirement as it does not use a star schema, placing it closer to the workload schema in TPC-C.

The mixed workload in HTAPBench uses all the entities in TPC-C and TPC-H's `Nation`, `Region` and `Supplier`, as proposed in [Cole et al., 2011]. The remaining TPC-H entities were merged in a non-intrusive way into TPC-C's workload. The result is a workload that matches Gartner's recommendations for hybrid workloads, where data should not be moved from operational to data warehouses in order to support analytics, but live under the same schema allowing drill-down analytical operations to point toward the freshest data produced by the operational activity [Pezzini et al., 2014].

The OLTP execution in HTAPBench runs according to a target number of transactions per second (*tps*). It is thus necessary to ensure the optimal configuration regarding some TPC-C specific parameters such as the total number of warehouses and clients, defining the number of transactions per minute (*tpmC*).

To compute these parameters, we refer to the TPC-C specification [Council, 2010a] and use the characterization for the TPC-C's ideal client, considering the minimum think time for each transaction type, and provided that transactions do not fail, no rollback operations. According to TPC-C, a single client should not be able to execute more than 1.286 *tpmC*. Under these conditions, it is possible to extract the target *tpmC* from the target *tps* (3.1), as well as the total number of *clients* (3.2) and *warehouses* (3.3). The required target *tps* is one of the configurable criteria in HTAPBench and directly relates with the expected scalability of the system and respective database size (further details are provided in subsection 3.3.2).

$$target(tpmC) = target(tps) \times 60 \times \frac{\%NewOrder}{100} \tag{3.1}$$

$$\#clients = \frac{target(tpmC)}{1.286} \tag{3.2}$$

$$\#warehouses = \frac{\#clients}{10} \tag{3.3}$$

The business queries in TPC-H are built from filtering, aggregation and grouping operations over a given result set. Filtering operations use SQL operators such as `where`, `having` or `between`. Their main goal is to limit the number of considered rows. Since the transactional activity will feed the analytical queries in HTAPBench, the number of filtered rows will grow over time. If not safeguarded, the results of these analytical queries are poised to become incomparable across runs. On the one hand, data distributions regulate how the parameters for filtering operators are selected, enabling the queries to dynamically exercise several regions of the dataset while exhibiting comparable complexity. On the other hand, if the queries are not dynamically generated, the use of fixed bounds on the filters would end up traversing the full qualifier domains, preventing the query planner of the SUT to be exercised.

To verify the impact from using fixed or dynamic parameters, we conducted an experiment where we considered the execution of the 22 TPC-H queries over 2 setups. The first used a set of fixed parameters that would resemble full domain searches. The second considered dynamically generated parameters that were not bound to a particular data distribution. The configuration of dynamically generated parameters created a new set for each run, while the fixed configuration reuses the same set across runs. Each setup considers the average of 5 independent executions. Queries were computed against a column-oriented engine. In each run, the database was populated with one warehouse and the queries were executed without any of the filtering operators in their composition, establishing a baseline comparison that represents the universe of rows in each query.

The experiment observes the average difference of result set row count in consecutive executions of each run, for a given setup, as a percentage of the baseline result. When considering fixed parametrization, the result set cardinality did not change across consecutive runs, and in most cases, queries ended up selecting a considerably broader space of tuples. When we used dynamically generated filters in the TPC-H queries, a variation of up to 77% in result set cardinality was observed in comparison with the baseline. This was due to not using a distribution to feed the date fields during the population stage of the benchmark. By not using a distribution to regulate how these fields are generated, it becomes likely that the items inserted during the populate stage of the benchmark present uneven time distributions when compared with the ones created during the execution stage. The next section introduces a way to generate a workload distribution that ensures analytical queries with comparable complexity across runs.

3.2 Result Set Homogeneity

The analytical queries composing a hybrid workload are fed with data created or manipulated by the transactional agent, either during the initial populate stage, or during the transactional execution part of the hybrid workload. The engine qualifying as HTAP should operate under an isolation criterion that enables the analytical queries to observe data committed by the transactional agent at the time the analytical queries started. Likewise, a given analytical query should freely access the entire dataset, spanning from the first to the last committed transaction.

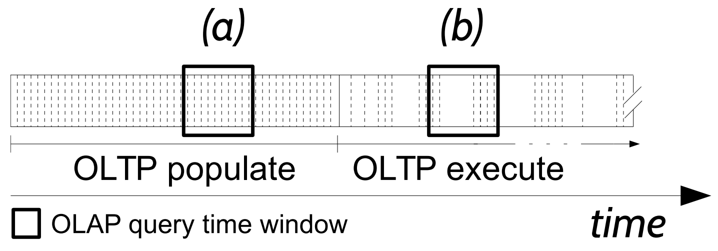


Figure 3.3: Timestamp density difference.

As discussed in the previous section, the absence of a regulating mechanism would result in the use of randomized query boundaries, producing incomparable results across runs. The same may happen when analytical queries observe data generated in the populate and execution stages of the hybrid workload. Figure 3.3 depicts an example of the patterns where data is created or changed by the transactional agent. On the one hand, the OLTP populate stage (Figure 3.3(a)) promotes bursts of transactions inserting data, causing a high concentration of timestamps in a short time period. On the other hand, during the OLTP execution stage, the OLTP transaction rate is regulated by TPC-C.

What is desirable is that the pattern generated by the OLTP execution within TPC-C is also observed by analytical queries whenever they traverse the data loaded during the populate stage. To mitigate this issue, we introduce a density extraction mechanism that ensures the same data pattern across stages. Briefly stated, our approach observes the amount of generated date fields during the execution stage of TPC-C, allowing the system to apply the extracted density during population.

Density Function

The populate and execution stages of TPC-C generate different date densities across the whole dataset, varying according to the configured transaction mix within TPC-C. Moreover, as not all transaction types generate the same number of new timestamps, we configured our density function to reflect that behavior.

$$txnMix = \frac{\%NewOrder + \%Payment + 10 \times \%Delivery}{100} \tag{3.4}$$

$$d(T_S/s) = tps \times txnMix \tag{3.5}$$

Both the `New Order` and `Payment` transactions generate one timestamp each, while each `Delivery` transaction generates ten. The `Order Status` and the `Stock Level` transactions do not generate any timestamps. The amount of generated timestamps is a direct consequence of the complexity associated with each transaction. A transaction that considers more timestamps introduces more modifications in fields of the workload schema that directly impact the density observed. It is then possible to express density as a function between the target number of transactions per second and the ratio of `New Order`, `Payment` and `Delivery` transactions, as defined by (3.5). In the following, we set up an experiment that allowed us to observe the expected density.

tpmC	clients	warehouses
635	495	49
741	576	58
886	689	69

Table 3.1: Workload configuration - ideal TPC-C client.

This experiment was conducted on a server with an Intel Xeon x3220 2.4 *GHz* QuadCore, 8GB of memory and 128GB Solid State Drive. For the purpose of this test, we relied on a hybrid system. The configurations considered are depicted in table 3.1, reflecting workloads with more than 70GB in total size. In each experiment, the database was dropped and populated. Afterwards, we ran TPC-C under the standard transaction mix in runs that lasted 60 minutes.

The results depicted in table 3.2 are the average of 5 independent runs regarding each target. The results depict an increasing amount of newly issued timestamps (T) as the defined target increases, thus reflecting a density function that also presents an increasing trend.

The results also show that the density function provides results that are only 3% apart when comparing with the experimental observation.

The timestamp density will introduce a change in the standard TPC-C specification. It is worth noting that this modification does not introduce any change in TPC-C business logic. The individual TPC-C results are kept comparable with a same-sized TPC-C installation.

tpmC	Total Observed (Ts)	Expected $d(T_s/s)$	Experimental $d(T_s/s)$
635	108,051	30.24	30.01
741	125,500	35.14	34.86
886	150,114	42.02	41.69

Table 3.2: Density observation results.

3.3 HTAPBench Metric and Components

Unified Metric

The disparity in workload complexity and structure between OLTP and OLAP workloads is a major hurdle when trying to define a unified metric for a HTAP system. So far, one of the main disadvantages of previous approaches was the fact that they would enable both OLTP and OLAP executions to grow in order to achieve the maximum throughput for each. The non-regulated growth induced by OLTP execution would inherently degrade OLAP performance since analytical queries would have to scan more data. HTAPBench removes one axis of variability by regulating the OLTP workload. The database size is key to measure the complexity of OLAP queries, as they commonly traverse full data domains, directly causing an impact on data acquisition. The assurance of a constant transactional execution leads to a sustained and known database growth; that is, the rate at which OLAP queries observe new data is fixed and predictable, bypassing the need to normalize the OLAP results in terms of the observed growth. In a scenario of unknown growth rate of the database, analytical results would have to be normalized in order to be comparable across runs.

$$QpHpW = \frac{QphH}{\#OLAPworkers}@tpmC \tag{3.6}$$

Expression 3.6 defines the metric we propose, *QpHpW* or *Queries of type H per Hour, per Worker*. It reads as the number of analytical queries executed per OLAP worker. In a system that is able to sustain the configured *tps*. It is defined by the ratio between TPC-H's metric and the total number of registered OLAP workers induced by the client balancer. A higher *QpHpW* corresponds to a system that is able to compute more queries per analytical worker, thus representing a higher overall analytical throughput. Achieving the best configuration for a given installation becomes a multi-run optimization problem regarding a given target *tps*.

Client Balancer

The client balancer module is responsible for monitoring and deciding whether or not to launch additional OLAP workers. When the OLTP agent ensures that the target *tps* is stable, the client balancer will periodically assess whether or not the SUT is capable of handling an extra OLAP worker. This assessment relies on a proportional-integral feedback controller.

$$output = K_P \Delta tps + K_I \int \Delta t \tag{3.7}$$

The feedback control mechanism (3.7) is characterized by a proportional (K_P) and an integral (K_I) gain adjustment. The gain parameters are used over the found system deviation ($\Delta tps = target\_tps - measured\_tps$) to compute the output value. The correct adjustment of either gain factor is vital to ensure that the feedback controller does not exceed too quickly the SUT's capabilities, which in turn would launch a higher number of OLAP workers, causing disruption on the throughput of the OLTP execution. We tuned the proportional-integral feedback controller by experimentation to use the gain adjustment characterized by $K_P = 0.4$ and $k_I = 0.03$. Algorithm 1 presents the client balancer decision process.

The client balancer will periodically (Δt) poll the engine regarding the current number of transactions being delivered. This information is then fed into the feedback controller. To launch another OLAP worker, the client balancer ensures that the output value produced by the feedback mechanism is within a given threshold of the configured target *tps* and that the system is not saturated. The point of saturation is reached when the current number of OLTP transactions per second being delivered drops below the chosen threshold.

Density Consultant & Loader

HTAPBench follows the standard TPC-C transaction mix. The density consultant computes the correct density according to the chosen target *tps* and trans-

Algorithm 1 Client Balancer

```
1: procedure
2:     $wait(\Delta t)$
3:     $threshold \leftarrow target\_tps - measured\_tps$
4:     $integral \leftarrow integral + threshold \times \frac{1}{\Delta t}$
5:     $output \leftarrow K_P \times threshold + K_I \times integral$
6:     $previous\_threshold \leftarrow threshold$
7:
8:     if $output > (target\_tps \times margin)$ and $\neg saturated$ then
9:         $start\ OLAP\ worker$
10:    else
11:        $saturated \leftarrow true$
12:    end if
13: end procedure
```

action mix. During the populate stage, in order to generate timestamps that follow the required density, the loader component is equipped with a clock that initiates with the system time at which the populate stage is initiated. The clock then computes how much time should elapse between clock ticks (ΔT_S) in order to fulfill the required density, as defined in (3.8).

$$\Delta T_S(ms) = \frac{1}{d(T_S/s)} \times 1000 \tag{3.8}$$

The HTAPBench loader will proceed to create and load all the table entities represented in the hybrid data schema, built from merging TPC-H's schema into TPC-C's. The final installation will scale in size according to the computed number of warehouses. When loading tables with references to date items, the load worker makes use of the clock, increasing one tick for each new date field to be loaded. After completion, applying the density function ensures that the temporal density in the date fields matches the observed density during execution of the transactional workload.

Dynamic Query-H Generator

The analytical queries within HTAPBench are constructed according to the TPC-H specification, which requires them to be built with randomized parameters within given boundaries. The dynamic query-H generator module is responsible for building the SQL statements for the queries, ensuring that the random values

```
select sum(ol_amount) as revenue from order_line from order_line
    where ol_delivery_d between [Date] and [Date+1 year] and
        ol_quantity between [ammount a] and [amount b]
```

Figure 3.4: TPC-H query 6.

comply with the TPC-H specification. This module integrates with the client balancer module that will launch the analytical workers, with its output afterwards fed into the TPC-H worker. The dynamic query-H generator computes the window frames that should be considered for query execution. It dynamically adjusts the window frame generation to reflect a sliding window behavior to include items generated by the transactional activity. This contrasts with the static time frames specified in TPC-H. Take as an example query `Q6`, that computes the total revenue for orders placed within a given period. This particular query restricts the result set to orders placed within a one-year time frame, starting on January the first of a randomly selected year between 1993 and 1997, and ending in the following year. In TPC-H, this is possible since it does not consider database growth. However, in HTAPBench, the database grows at the pace dictated by the OLTP execution of the benchmark. Thus, if window frames were to be kept static, the new regions on the dataset would never be queried. To produce a homogeneous result set that is representative of the whole dataset, the dynamic query-H generator ensures that queries comply with the time range imposed by the specification while simultaneously leveraging the density consultant module to shift the starting date of the range to meet the speed at which the OLTP execution is making the database grow. Hence, the sliding window behavior not only ensures that the entire dataset is considered, but also, that consecutive executions of the same query are kept comparable.

Results Monitor

The results monitor collects the execution results produced by each worker. The final measurements are only collected after the configured execution time elapsed and all the workers finalized all procedures. The transactional activity is characterized according to TPC-C's metric (*tpmC*) while, the analytical activity is characterized by TPC-H's metric (*QphH*). Together with both metrics, this module also outputs data from the client balancer, characterizing each run in terms of how many OLAP workers were launched, when they were launched and the result set volume and latency for each analytical query executed.

3.3.1 Implementation

HTAPBench is available as an open source project.[1] It was implemented in Java for improved portability and includes all the aforementioned components. The current prototype was implemented as an extension of OLTPBench [Difallah et al., 2013], a framework that enables the execution of several benchmarking approaches. However, OLTPBench's implementation of TPC-C does not consider the think time used during transaction execution. The TPC-C specification requires a time in which each transaction simulates (by entering a sleep stage) the time required by the terminal user to insert data, as well as the terminal's processing time (TPC-C's simulation of a real-world scenario). Consequently, as part of our extension to OLTPBench to implement HTAPBench, we introduced the think time processing stage for each transaction of TPC-C.

HTAPBench relies on Java Database Connectivity (JDBC) to establish a connection with the SUT. Since JDBC defines a standard interface to connect with several engines, it is possible to use HTAPBench with a wide range of engines.

3.3.2 Benchmark Configuration

HTAPBench has several system requirements. The machine running HTAPBench should be provisioned with a Java distribution and the appropriate database engine driver.

HTAPBench requires the user to provide the engine JDBC connection URL, the fully qualified name of engine driver class and the target number of transactions per second (*tps*). The required target *tps* for a given configuration derives from an expectation regarding the performance compliance for the SUT. The user should start with a small target *tps* value and progressively increase the target *tps* until the SUT is saturated, as it happens with the warehouse scalability in TPC-C. Other configurations are allowed, enabling customization of the workload, namely: the client balancer Δt and `throughput threshold`, the

[1]https://github.com/faclc4/HTAPBench.git

`TPC-C transaction mix`, the `TPC-H query mix` and the `execution time`. We briefly describe the impact of these mandatory parameters on the system.

- `Client Balancer` Δt: This parameter configures the period in seconds used by the client balancer module to assess if further OLAP workers should be launched. On the one hand, assigning the appropriate evaluation period has a direct impact on the convergence time and precision of the benchmark. Choosing a small value may cause the system to converge too quickly and overestimate the number of admissible OLAP clients. On the other hand, choosing a large value improves the client balancer decision by exposing it to a larger number of samples, but delays the overall process. This parameter defaults to 60 seconds.

- `Client Balancer throughput threshold`: This parameter configures how sensitive to change the client balancer should be, setting up the range of allowed *tps* computed in percentage of the target *tps*. It has a default value of 20%, which by experimentation we consider to be a reasonable trade-off of loss OLTP throughput in favor of having OLAP capability.

- `Execution time`: This parameter configures the duration of time that each run of the execution stage of the benchmark should last. This parameter defaults to 60 minutes, configuring a reasonable execution time after the warmup stage for the client balancer to exercise the SUT.

The execution of HTAPBench is straightforward. After system configuration, the user should run the populate script that generates the appropriate Comma-Separated-Values (CSV) files for the configured *tps*. Afterwards, the user should consult the documentation of the engine to be tested in order to use the correct procedure to load the CSV files (optionally, it is possible to directly populate the SUT, but this operation usually takes longer, since it does not load data in batch mode). To start the execution stage, which includes the initial warmup, the user should use the execution script, automatically deploying the required number of clients. The execution will run for the configured time and will afterwards produce result files, characterizing the OLTP and OLAP executions and computing the unified metric.

3.4 Benchmarking Campaign and Validation

The benchmarking examples presented in this section result from an extensive study intended to evaluate the key properties of HTAPBench. As such, the main purpose of the presented scenarios is not to quantitatively compare different SUTs, but rather to demonstrate the expressiveness of the benchmark suite and its metrics.

Three different SUT were selected, namely: (*i*) an OLTP system, (*ii*) an OLAP system and (*iii*) a HTAP system. Besides the target *tps* and the client balancer throughput threshold, the same HTAPBench configuration was used across experiments. The client balancer was set to use an evaluation period (Δt) of 60 seconds and the OLTP activity was regulated by the standard transaction mix within TPC-C. For the OLAP activity, we set up HTAPBench so that each business query would be chosen according to a uniform distribution. Across experiments, we configured HTAPBench to inject 100 transactions per second, which according to our density extraction mechanism amounts to 2,099 active OLTP clients and 210 warehouses, rendering a dataset with a total of 117GB of data. The selected target *tps* was chosen as the number of configured warehouses generate a dataset with over 100GB, which is above the third recommended scale factor for the OLAP agent (*e.g.,* 1GB, 10GB, 100GB). All the following experiments reflect the average of 5 independent 60 minute runs.

3.4.1 OLTP System

The current experiment used a server with an Intel Xeon E5-2670 v3 CPU with 24 physical cores (48 virtual) at 2.3 GHz and 72GB of memory, running Ubuntu 12.04 LTS as the operating system. We deployed an OLTP row-oriented engine and configured the level of memory that would allow the required number of clients. The client balancer within HTAPBench was configured to consider the default throughput threshold of 20%. The admissible loss of OLTP throughput induced by the configuration of the threshold is depicted as a gray area in figures 3.5, 3.6 and 3.7. HTAPBench was launched from the same machine.

The results in Figure 3.5 depict that HTAPBench was able to launch and sustain the required target *tps* throughout the entire execution time. This can be read from the OLTP line that is the linear interpolation of the plotted points. The required target *tps* was reached on the first minute of execution and, from that moment on, the client balancer started its configuration by launching an OLAP worker at each minute. The evolution regarding when and how many OLAP workers exist is read by looking at the plotted line resembling a staircase.

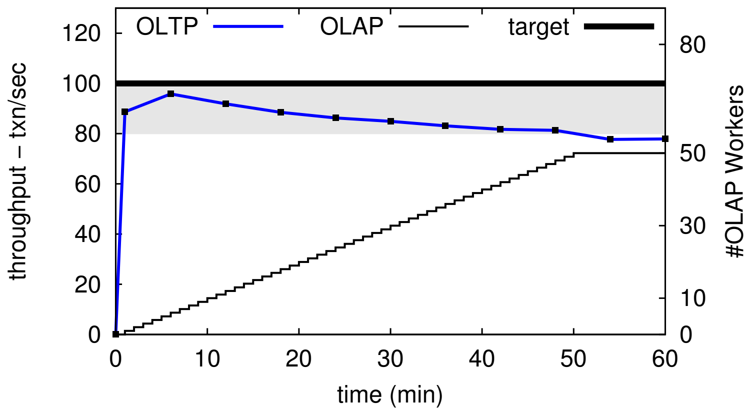


Figure 3.5: OLTP SUT

With a configured throughput threshold of 20%, the bottom *tps* barrier was only surpassed after 50 minutes, saturating the system at that point in time, and therefore not launching further OLAP workers. By default HTAPBench's client balancer introduces a single OLAP worker in each evaluation period (Δt). It may however be configured to deploy more than one OLAP worker at time. For the current experiments, in order to understand the impact in the OLTP throughput from increasing the OLAP activity with a higher granularity, we deployed only a single OLAP worker at a time.

Throughout the test, a declining trend in the OLTP throughput becomes evident. In what strictly concerns the OLAP activity, the engine under test was able to hold up to 50 OLAP clients. As previously stated, the main goal in HTAPBench's evaluation scheme is to discover how many OLAP workers are required to stress out a given SUT, while ensuring that the transactional activity does not degrade beyond a configurable threshold. In addition to the temporal evaluation of both OLTP and OLAP workers within the system, HTAPBench outputs the unified metric. Within this setup, the SUT was able to sustain 756 *tpmC* and 7 *QphH*, resulting in 0.14 QpH in each OLAP worker (7 QpH / 50 OLAP workers). As such, the unified metric, *QpHpW*, amounts to 0.14 @ 756 *tpmC*.

3.4.2 OLAP System

For this experiment reused the previous configuration. We deployed an OLAP column-oriented engine, only setting up the required level of clients allowed in the engine.

The results in Figure 3.6 depict that the SUT sustained the OLTP throughput for a shorter period. This behavior is not surprising since the focus of this engine is not on OLTP activity. From the moment the threshold was broken (6th minute), the client balancer stopped triggering new OLAP clients. From this time on, albeit at a lower throughput, the OLTP activity was kept stable until the end of the run time. The SUT was able to register 217 *tpmC* while sustaining 4 individual OLAP clients. Cumulatively, the OLAP activity reached a peak of 123 *QphH*. As such, the unified metric we propose, *QpHpW*, reaches 30.75@217 *tpmC*.

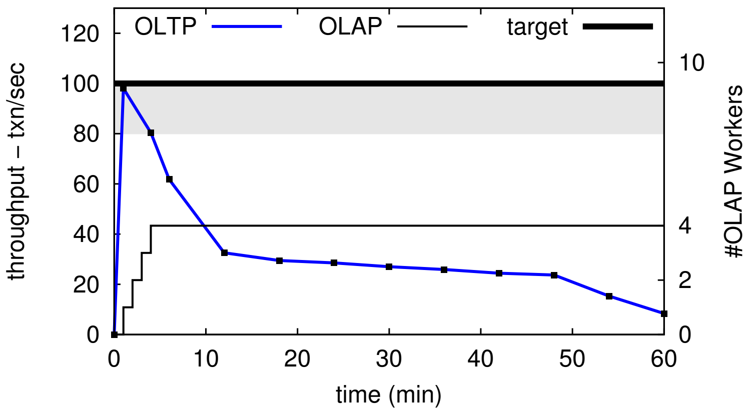


Figure 3.6: OLAP SUT

Compared to the previous system, it can sustain a larger number of analytic queries, despite the fact that this particular system was only able to launch 4 OLAP streams, compared to 50 OLAP streams with the first SUT. This may seem counter-intuitive, considering that we are working atop a column-based engine specifically designed for an analytical workload. However, the 4 OLAP streams in the current SUT are much more efficient when compared with the 50 OLAP streams in the OLTP SUT.

3.4.3 Hybrid System

Next, we use HTAPBench to characterize a HTAP engine. As such systems are designed to scale out, they are usually made available over a distributed architecture.

We deployed this system over 10 nodes, 9 of which are responsible for handling and storing data, and the remaining node provides coordination and other global services. Each node has an Intel i3-2100-3.1GHz 64 bit processor with 2 physical cores (4 virtual), 8GB of RAM memory and 1 SATA II (3.0Gbit/s) hard drive, running Ubuntu 12.04 LTS as the operating system and interconnected by a switched Gigabit Ethernet network.

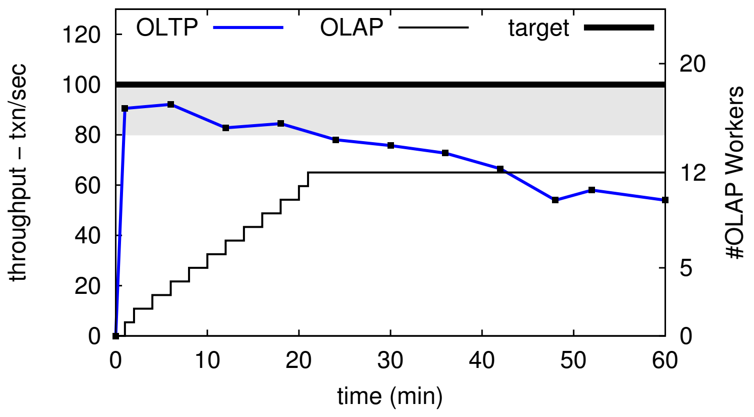


Figure 3.7: Hybrid SUT

The results depicted in Figure 3.7 reveal that the OLTP throughput reached the assigned target in the first minute of execution. From the first minute onward, the client balancer started to deploy OLAP streams until the OLTP throughput degraded beyond the considered threshold, which happened after 20 minutes, registering a maximum of 12 OLAP streams. From that moment on, the OLTP throughput slowly degraded over the remainder of the test. Cumulatively, the SUT was able to sustain 530 *tpmC* and 169 QphH. Therefore, our unified metric *QpHpW*, evaluates as 14.14@530 *tpmC*.

3.4.4 Discussion of Results

The results of these three tests are summarized in table 3.3. Although the OLTP SUT achieved the highest number of OLAP workers, the results in Figure 3.3 also

	# OLAP workers	QpH	QpHpW
OLTP	50	7	0.14 @ 756
OLAP	4	123	30.75 @ 217
Hybrid	12	169	14.14 @ 530

Table 3.3: Analytical results according to distinct workload profiles.

show that it achieved the lowest OLAP performance (0.14 QpHpW). On the one hand, the OLAP workers in the OLTP engine spend most of their time waiting for the OLAP queries to be processed and therefore complete relatively few OLAP queries. On the other hand, the OLAP engine processes significantly more analytical queries, but fails to cope with the required OLTP throughput that is used by the client balancer to launch additional OLAP workers. Overall the OLTP engine completed 7 QpH and the OLAP engine completed 123 QpH. This results show that the OLAP SUT was better at handling the OLAP workload, but its inability to cope with the OLTP workload in the hybrid configuration harmed the overall scalability.

The ideal hardware configuration for the Hybrid SUT is significantly different from the OLTP system or the OLAP system, which prevents a direct comparison among them. Nevertheless, HTAPBench was able to evaluate the Hybrid SUT, enabling 12 OLAP streams and achieving a total of 169 OLAP queries. The results reveal that the Hybrid SUT can sustain a considerable OLTP throughput with a moderate OLAP scalability.

3.5 Validation

In order to validate the expressiveness of our metrics, we studied the system's representativeness, workload accuracy, homogeneity, throughput threshold variability and cost. Moreover, we discuss the benchmark portability, reproducibility and repeatability. For that matter, we used all the previous SUT configurations.

First, we show that the proposed metric can both identify systems with similar or very different goals. Second, we analyze HTAPBench's accuracy by verifying the storage traces produced in three distinct scenarios. Third, we verify that the produced query result sets are homogeneous. Fourth, we conduct a variability analysis which presents the consequences of varying HTAPBench's throughput threshold. Finally we discuss the cost of using the suite.

3.5.1 Unified Metric

The unified metric allows us to establish a comparison across two dimensions. Figure 3.8 uses a quadrant field plot to visually compare the relationship between *QpHpW* and the target OLTP throughput.

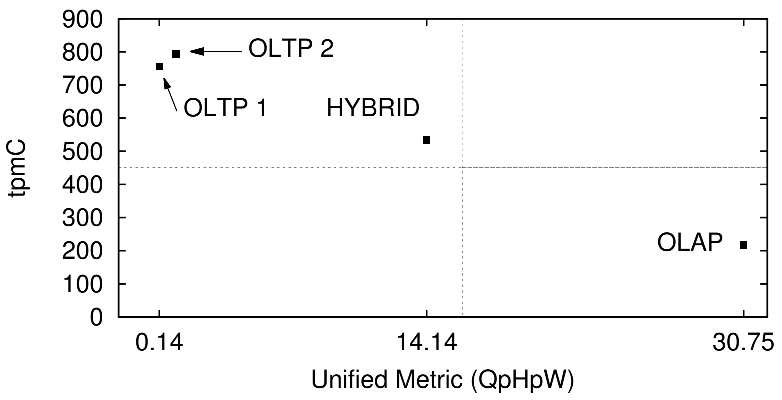


Figure 3.8: Quadrant field plot for the unified metric.

While an increase in the vertical axis translates into a higher OLTP throughput, the horizontal axis evaluates the capability to perform more analytical queries per OLAP worker. The best result would be a reading on the upper right quadrant. The analysis depicts that while the SUT labeled OLTP 1 registered a *QpHpW* of 0.14@756 *tpmC*, the OLAP observed 30.75@217 *tmpC* and the hybrid 14.14@534.1 *tpmC*. Overall, we can state that the analyzed systems follow the trend in which the increase of OLTP activity diminishes the OLAP capability. The hybrid system reached a position close to the middle of the plot, indicating better support for the mixed workload with simultaneous OLTP and OLAP activity.

So far, the presented results allow us to conclude that the benchmarking suite is able to compare systems with very distinct work plans. In order to verify if the suite is able to distinguish systems designed for similar workloads, we reproduced the same experimental scenario as in subsection 3.4.1 but changed the SUT to a different engine designed for OLTP workloads. We evaluated its performance and plotted it in Figure 3.8 with the label "OLTP 2". The results place OLTP 2 very close to system OLTP 1 in the same quadrant, which indicates that both systems must belong to the same class (*i.e.,* OLTP, OLAP or HTAP).

All of the previous experiments were integrated in a statistical study from where we were able to compute the variation coefficient of the computed metrics, pointing to a variation of 1.2%.

3.5.2 Throughput Threshold Variability

The client balancer in HTAPBench controls whether further OLAP streams are deployed or not, evaluating at each point in time if the current OLTP throughput does not go below a threshold defined by the throughput threshold configuration. Intuition leads us to believe that by increasing the throughput threshold, the client balancer will naturally allow more OLAP clients to be deployed, thus increasing the overall amount of analytical queries performed.

Throughput threshold	*QpH*	*tpmC*	*OLAP Clients*
10 %	70.99	130.64	1
20 %	168.9	131.36	5
40 %	265.99	138.30	11

Table 3.4: Client balancer throughput threshold variance.

The current experiment assesses if in fact such behavior translated into the actual behavior of the benchmark. For this purpose, we set up the OLAP SUT as in subsection 3.4.2 but varied the throughput threshold across runs.

First, by analyzing table 3.4, the reader shall notice that the consecutive executions registered similar OLTP throughputs as expected. Moreover, as we increase the allowed threshold, we verify that more analytical queries were performed, which is a consequence of having more OLAP clients on the system, thus increasing the registered QpH and consequently the QpHpW.

3.5.3 Workload Representativeness

Database systems must be evaluated with representative workloads that test realistically the storage back-end capabilities. To better understand the representativeness of HTAPBench we analyzed the storage traces produced. The 3 previously tested SUT were evaluated with these workloads and the resulted storage traces were collected and analyzed with the *blktrace* tool.[2] This tool allows us to collect storage traces for a given block device. With these traces it is possible to extract useful information, such as the access patterns of storage requests, the ratio of storage reads and writes, the throughput and latency of each request, *etc.* Each SUT was deployed in a single machine as in subsection 3.4.1.

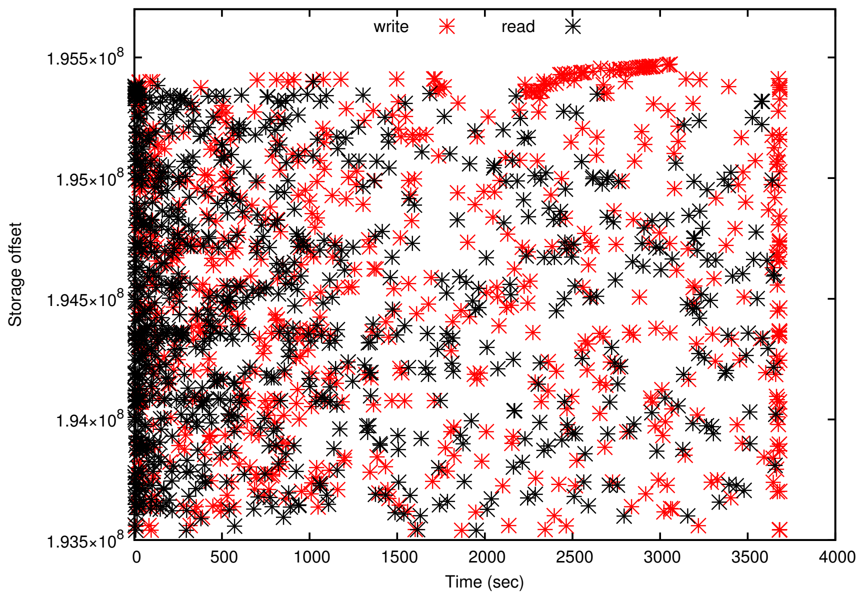


Figure 3.9: Disk access pattern registered in a solo OLTP workload.

Figures 3.9, 3.10 and 3.11 depict the access patterns registered by the storage medium for an HTAP SUT. The figures plot the offset of the storage medium (vertical axis) being accessed during execution time of the benchmark (horizontal axis). Black and red marks respectively depict read and write operations. The analysis of the traces collected allowed us to confirm that; on the one hand OLTP workloads are dominated by random storage accesses (Figure 3.9) while OLAP workloads do mostly sequential storage accesses (Figure 3.10) dominated by read-

[2]blktrace manual: `http://linux.die.net/man/8/blktrace`.

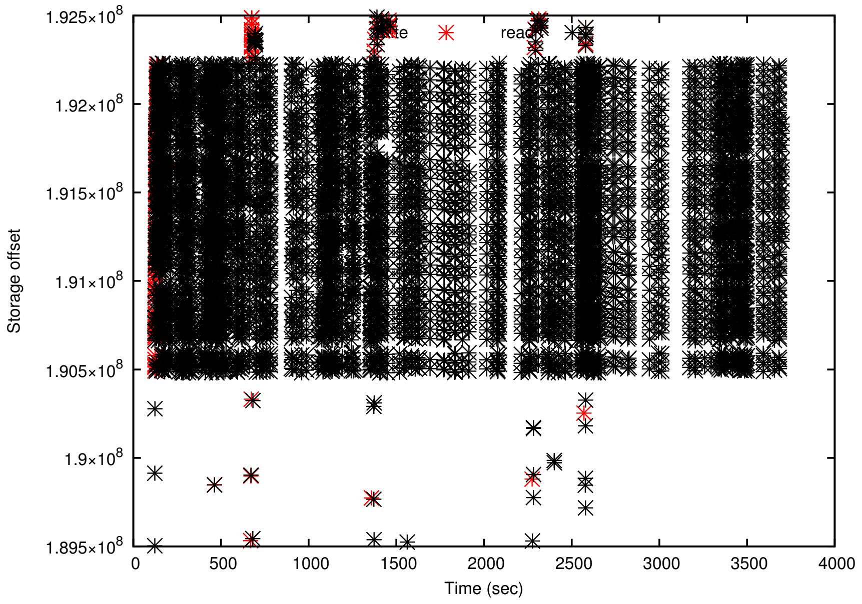


Figure 3.10: Disk access pattern registered in a solo OLAP workload.

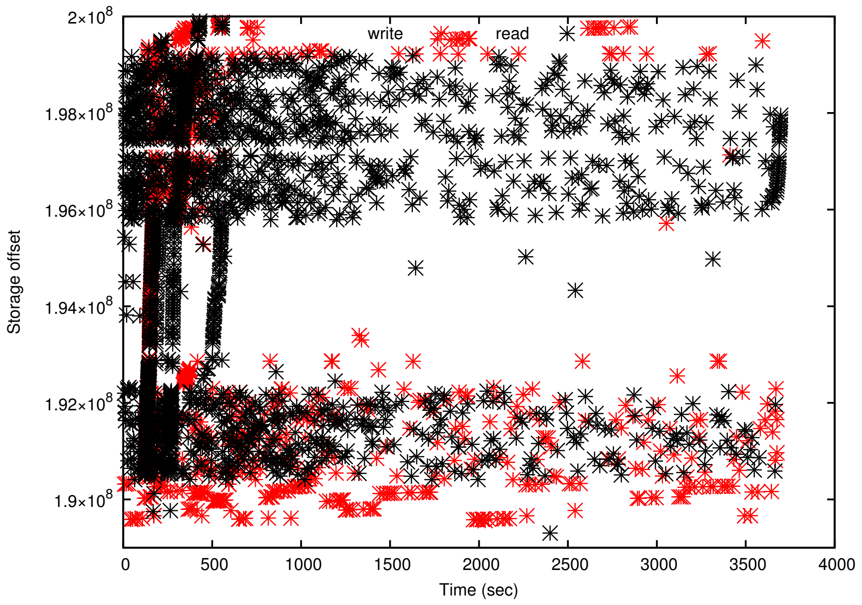


Figure 3.11: Disk access pattern registered in a hybrid workload.

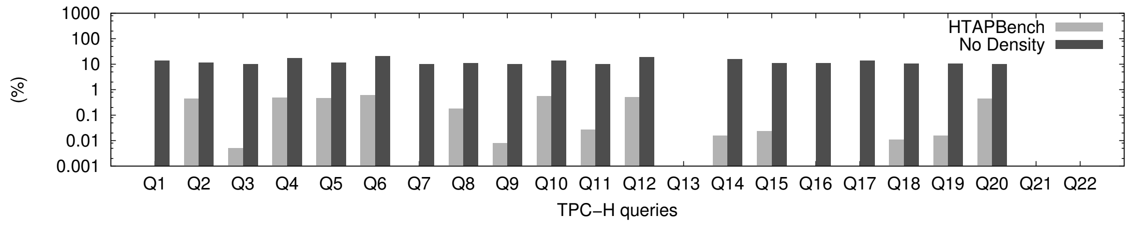


Figure 3.12: Result set execution cost as a percentage of the baseline across independent runs. The vertical axis is presented in logarithmic scale. The absence of columns represents null variability.

only requests. On the other hand, mixed workloads generated by HTAPBench present a mix of these access patterns, as expected (Figure 3.11). Moreover, the ratio of storage reads and writes is according to the specification of each workload.

In summary, these results show that HTAPBench is able to simulate a realistic storage load for a hybrid SUT that process both OLAP and OLTP workloads simultaneously. Consequently, HTAPBench's representativeness of the analytical capability on top of a sustained transactional operation is ensured by the experimental data, but also by the individual representativeness of the TPC workloads used. Moreover, the results show that current proposals for hybrid SUT must be aware that processing simultaneously OLAP and OLTP workloads generates a mix of random and sequential storage accesses. This challenge can drive novel research proposals for both hybrid engines and the back-end storage systems supporting them.

3.5.4 Homogeneity and Reproducibility

The present experiment validates the effectiveness of our density mechanism. Figure 3.12 depicts 2 different scenarios, where we compare a distribution using random field generation with the density approach we propose. The depicted results relate to a baseline comparison where all queries were executed without any of their filtering operators, thus allowing us to extrapolate the total universe of rows considered in each analytical query. The value presented for each query is the absolute difference in produced result set rows in two consecutive executions: first, with random parameters that do not follow any distribution and, second, with the density mechanism proposed in HTAPBench.

In the first scenario, the analytical queries ran with the introduction of randomized parameters. The results show that consecutive executions of such queries produced variable result sets. The measured variability of result set lines across all query executions for this setup amounted to 12%, with a registered maximum of 20%. For the second scenario, we used HTAPBench's density mechanism to generate the random parameters to be used in all analytical queries. Queries Q13, Q21 and Q22 produce variabilities of 0% as the consecutive runs always produced the same number of result set rows.

The results show that by using our approach, we were able to produce query result sets with comparable costs. Consecutive executions of the same query registered a variability of just 0.17% on average across the different SUT and different configuration settings. These results confirm a very high level of reproducibility of results for a given configuration.

3.6 Related Work

Several organizations proposed benchmarking standards to assess either transactional or analytical systems, namely: the Transaction Performance Council (TPC) [Council, 2015c], the Standard Performance Evaluation Council (SPEC) [Council, 2015a] and the Storage Performance Council (SPC) [Council, 2015b]. Following, several systems are categorized into OLTP, OLAP or Hybrid and a brief presentation of their main characteristics and shortcomings is provided.

Benchmarks for Transactional Systems

Online Transactional Benchmarking systems, such as TPC-C [Council, 2010a], focus on assessing a system's ability to cope with a large amount of small-sized transactions. Usually, OLTP systems rely on row-oriented data stores, in which the transactions operate over a restricted space of tuples, ensuring at the same time properties such as consistency and isolation in what is commonly referred to as ACID [Haerder and Reuter, 1983].

The TPC-C specification models a real-world scenario where a company, comprised of several warehouses and districts, processes orders placed by clients. The workload is defined over 9 tables operated by a transaction mix comprised of five different transactions, namely: New Order, Payment, Order-Status, Delivery and

Stock-Level. Each transaction is composed of several read and update operations, where 92% are update operations, which characterizes this as a write heavy workload. The benchmark is divided into a load and an execution stage. During the first stage, the database tables are populated and, during the second stage, the transaction mix is executed over that dataset. TPC-C defines how these tables are populated and also defines their size and scaling requirements, which is indexed to the number of configured warehouses in the system. The outcome of this benchmark is a metric defined as the maximum qualified throughput of the system, *tpmC*, or the number of New Order transactions per minute.

The TPC-E [Council, 2015d] benchmark builds on TPC-C, introducing a variable number of client terminals. It models a scenario where a brokerage firm receives stock purchase requests from customers, trying to acquire the correspondent stock bonds from a stock pool. The purchase orders placed by clients are based on asynchronous transactions while stock requests between the brokerage firm and the stock exchange are based on synchronous transactions. Compared with TPC-C, this benchmark builds a much wider and more complex system as it is composed of 33 tables and 10 transactions, 6 of which are read-only and the remainder are read-write transactions; the latter accounting for 23% of all requests.

Both specifications build benchmarking suites strictly intended to evaluate OLTP engines. Despite their representativity of OLTP workloads, the characteristic short operational transactions prevent the workload from exercising OLAP requirements. Engines designed to provide a high OLTP throughput are typically row oriented as the nature of a single transaction induces I/O bound operations in several attributes per transaction. This behavior translates into random access patterns to the storage medium and does not produce CPU and memory bound operations as OLAP workloads do.

Benchmarks for Analytical Systems

Online Analytical Systems such as TPC-H [Council, 2010b, Poess and Floyd, 2000] or TPC-DS [Council, 2012] focus on assessing a system's ability to perform multi-dimensional operations, usually on top of column-oriented data stores.

TPC-H builds a workload that does not require ETL, modeling a real world-scenario where a wholesale supplier must perform deliveries worldwide. The business queries perform complex data operations (*e.g.,* aggregations) across large sets of data. The workload defines a query mix comprised of 22 business queries that access 8 different tables. The execution is divided into three stages. The first stage loads the database. During the second stage, a single user executes all 22 business queries, while during the third stage, a multi-user setup is used in order to evaluate the system's ability to schedule concurrent operations. TPC-H does not consider any growth factor during runtime, which means that the dataset does not change in terms of its total size. The outcome of this benchmark is computed through a metric that accounts for the total number of completed queries per hour (*QphH*).

TPC-DS builds a workload that requires ETL, particularly to ensure data freshness. It models a scenario where orders must be processed from physical and online stores of a wholesale supplier, mapping it into a star schema composed of 7 fact tables and 18 dimensions. The workload holds 4 different query types, namely: Reporting, Iterative, Data Mining and Ad-hoc queries. The database populated for TPC-DS, and as in TPC-H, it does not consider any growth factor; still, the initial population is regulated in terms of a scale factor that has direct influence over the data size. The output metric is defined as the number of queries per hour at a given scale factor, *QphDS@ScaleFactor*.

TPC-DS is seen as an evolution of TPC-H, addressing oversimplifications that prevent the proper evaluation of OLAP systems. However, the need to use ETL to promote updates on the star schema prevents us from using it as an OLAP agent in our hybrid workload.

The high prevalence of read operations in these workloads mostly generate sequential accesses to storage mediums, and therefore are not able to simulate the short and cross attribute nature of OLTP workloads that also live in a hybrid workload.

Benchmarks for Hybrid Systems

Hybrid workloads should capture both access patterns observed in the previous individual specifications [Funke et al., 2011]. There are a few benchmarking suites that use both access patterns, namely CH-benCHmark [Cole et al., 2011] and CBTR [Bog et al., 2011b,a, 2008, 2012].

CH-benCHmark creates a mixed workload also based on TPC standard benchmarks, enabling two types of clients. A transactional client provides a TPC-C agent, while an analytical client provides a TPC-H agent. To allow the analytical workload across the transactional relations, CH-benCHmark merged both schema into a single one, comprising relations from TPC-C and TPC-H. The relations accessed by the OLTP execution scale according to TPC-C's specification. Relations accessed by the OLAP execution are kept unchanged. However, CH-benCHmark neglects aspects within TPC-H's specification. Namely, the analytical queries should hold random parameters in order not to constantly transverse the same regions of the dataset. It also disregards the required distributions for date fields, impacting the produced analytical results.

CBTR defines a benchmarking system aimed at mixed workloads, which does not account for any previous standardized specifications, considering them too predictable. It introduces a workload built from real enterprise data that models an order-to-cash workflow. For that matter, a new schema and the respective transactional activity is presented. By using real data, CBTR bypasses the need to use numerical distributions to populate or to generate data during benchmark execution.

The major differentiator of HTAPBench when compared with the previous approaches lies specifically in its client balancer module that governs how both workloads coexist. Both CH-Benchmark and CBTR use naive approaches to find the maximum qualified throughput for both the transactional and analytical workloads. The main concern that arises is that neither workload agent in each benchmarking suite is aware of the other agent, creating a dispute for resources as each agent tries to saturate the SUT. The client balancer in HTAPBench follows Gartner's recommendation to specify how the transactional and analytical agents coexist; instructing the transactional agent to sustain a configured throughput and allowing the analytical agent to saturate the SUT up to the point when the transactional throughput is affected. Moreover, HTAPBench also addresses the issues found in CH-Benchmark relating to non-uniform result sets by using the density mechanism to regulate that behavior.

3.7 Remarks

This chapter focused on providing an insight on how different it is to assess hybrid database systems, introducing HTAPBench, a new benchmarking suite specifically engineered to evaluate hybrid database engines. The high levels of transactional activity and a simultaneous scalable performance, introduces new challenges to database systems targeted to solo OLTP or OLAP workloads, as demonstrated by the proposed unified metric. This new metric enables to quantify the analytical scalability factor observed while the transactional activity is kept within a configurable and bounded threshold. Moreover, the new client balancer orchestrates how both the transactional and analytical operation agents cooperate to assess the system's scalability.

HTAPBench introduced a mechanism that guarantees comparable results across runs, maintaining the expected randomness of the workloads but mitigating the previously observed complexity disparities. The action of the suite was validated on top of OLTP, OLAP and HTAP systems, demonstrating the expressiveness of the produced metrics. HTAPBench was able to distinguish different classes of systems (*e.g.,* OLTP from OLAP), as well as systems within the same class with high precision, while ensuring that the storage layout is exercised as expected for each workload type. Moreover, HTAPBench also contributed to validate the effectivenes of a distributed *shared-nothing* archicture to support a HTAP system.

Chapter 4

Distributed SQL Window Functions

Window Functions (WF) or OLAP Analytical Functions define a sub-set of analytical operations that allows the formulation of analytical queries in a derived view of a given relation. They were first introduced in the SQL:1999 standard as an optional feature but it was only in SQL:2011 that most database systems like Oracle [Corporation, 2015], IBM DB2 [IBM, 2013], Microsoft SQL Server [Coporation, 2013], SAP Hana [SAP, 2014], Cloudera Impala [Kornacker et al., 2015] or PostgreSQL [Postgresql, 2015] started offering the capability to interpret them.

WF enable a large set of useful analytical operators, offering a highly configurable environment together with a straightforward syntax. WF are particularly appealing for OLAP or hybrid workloads, allowing to compute time series analysis, cumulative sums, rankings, percentiles or aggregations over configurable logical data frames, customizing them to reflect newly ingested items from ETL or real-time production data. The benefits of this class of functions goes beyond windowed aggregates as they can also be considered for reducing both execution and syntactical complexity of other analytical queries [Zuzarte et al., 2003]. Ultimately, WF are frequently used in finance and science to provide trend and outlier analysis, which motivates their presence in around 10% of the queries in TPC-DS [Transaction Processing Performance Council, 2012], a benchmark suite built to evaluate data warehouse systems.

Database systems offer, to some extent, parallel capabilities that enable their analytical workloads to scale. Tipically these systems are designed to live in legacy-type servers that at most offer Many-Core architectures [Corporation, 2015, IBM, 2013, SAP, 2014]. Others combine this architecture with vast memory arrays for an in-memory only operation and fail to scale beyond the vertical scalability limit of the specific configuration.

A distributed configuration seeks to ensure modularity and better scalability, but this comes at the expense of increased coordination, resource costs or data partitioning. Particularly important for several classes of analytical functions such as WF is data partitioning. WF introduce the ability to compute moving aggregates over several configurable logical partitions of data. Parallel implementations of the WF environment need to account for the disparities between physical and logical partitioning of data in order to promote efficient executions. This requirement comes from the need to avoid unnecessary sorting of data in all stages of the execution, as it configures the costliest stage [Cao et al., 2012] in this environment.

This chapter focuses on WF, particularly leveraging their inherently parallel abstraction to expedite execution over a distributed layout. The building blocks of this class of analytical functions are addressed, providing an overview of the WF query construction and introducing the main properties of each clause that builds this environment. Data movement is particularly addressed, showing the cases where its mandatory action can be improved. For that matter, two statistical approaches are introduced to account for disparities found between physical and logical data layouts. First, a Holistic technique considers the logical partitioning induced by a query to guide the decision regarding which location should compute each section of a query plan. This is done by promoting intra-partition affinity of data. Second, a Similarity technique is introduced to exploit correlations between logical partitions, optimizing the required data movement between computing locations of the distributed architecture.

The remainder of this chapter is organized as follows: section 4.1 and section 4.2 address the building blocks of the WF environment. Section 4.3 particularly explores the impact that aggregations may have on the environment, while section 4.5 and 4.6 respectively cover the Holistic and Similarity mechanisms. Section 4.7 evaluates both techniques. Finally, section 4.8 reviews related work and section 4.9 discusses the achievements.

4.1 Window Functions

WF were introduced to enable complex analytical queries such as moving averages or cumulative sums to be easily expressed in SQL. Previously, some queries holding analytical functions were either impossible to express or required extensive joins and recursive computations, causing SQL queries to be extremely verbose [Zuzarte et al., 2003]. Unlike SQL, WF depend on the implicit or explicit order of tuples. The ordering independence in SQL introduces several optimization opportunities that may not be directly applied to WF. In fact, the ordering requirement is the costliest operation in the workload [Cao et al., 2012]. This introduces several challenges related to data partitioning in distributed approaches and the need for order. WF embody three concepts: the processing order, the partitioning of results and the notion of the current row being computed. They are expressed in SQL through the operator `OVER` and take the shape depicted in Figure 4.1.

```
select analytical_function() OVER( PARTITION BY A ORDER BY B ROWS
    BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) from R
```

Figure 4.1: SQL query with WF.

The understanding of the concepts in the upcoming sections is based on the following considerations:

- WF are computed after most statements (*e.g.,* `JOIN`, `WHERE`, `GROUP BY` or `HAVING`) but immediately before the final ordering (*i.e.,* `ORDER BY` outside the WF statement), if it exists;
- The result corresponding to the aggregation being computed will be added as an extra qualifier.

The produced results will not modify or filter the source relation, maintaining the cardinality of rows.

4.2 Window Function Query Construction

The WF environment can be decomposed into three stages, as depicted in Figure 4.2, defining the processing sequence: partitioning (1), ordering (2) and framing (3) stages. Each stage considers specific clauses, respectively: `PARTITION BY`, `ORDER BY` and `ROWS BETWEEN` or `RANGE BETWEEN`.

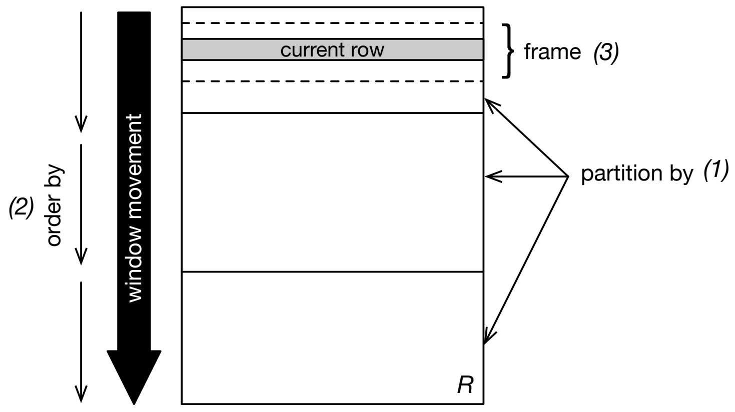


Figure 4.2: Stages of the Window operator: partitioning (1), ordering (2) and framing (3).

The environment combines different clauses, enabling the inclusion or exclusion of each clause type. For instance, it is possible to declare a WF with just a partitioning (*e.g.,* Figure 4.3) or ordering clause (*e.g.,* Figure 4.4). If no partitioning clause is declared, the entire relation is considered as a single partition. If no ordering clause is declared, then the natural ordering of the relation's key, or partitioning clause (if present) is considered. Moreover, each analytical function may implicitly influence computation logic set by the aforementioned clauses.

```
select analytical_function() OVER( PARTITION BY A ) from R
```

Figure 4.3: WF SQL query without ordering and framing clause.

```
select analytical_function() OVER( ORDER BY B ) from R
```

Figure 4.4: WF SQL query without partitioning or framing clause.

The next subsections analyze each part of the query construction.

4.2.1 Partitioning

The partitioning stage is defined by the `PARTITION BY` clause. It defines sets of distinct elements considered as logical partitions according to the qualifier or expression considered as argument for this clause. A single partition is built from each set of distinct elements in the qualifier defined as argument. The concept of partition will bound the analytical processing, meaning that the outcome of each analytical function will be restarted when a partition boundary is reached. If no partitioning clause is used, then the relation as a whole is considered.

The partition abstraction is particularly relevant for the intra-partition parallelism. That is, each logical partition becomes independent, allowing to deploy the processing of each logical partition to a single location, be it a CPU on a multi-core machine or a remote distributed computing instance.

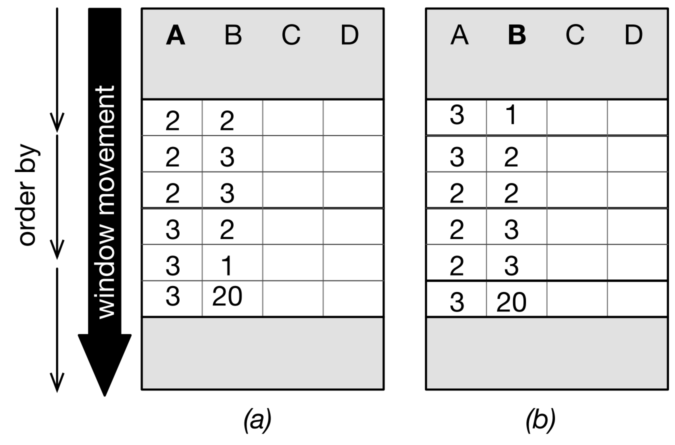


Figure 4.5: WF partition example. (*a*) Data sample *Partition by* qualifier A. (*b*) Data sample *Partition by* qualifier B.

Figure 4.5 depicts two data samples as the outcome of the WF query in Figure 4.3, but varying the qualifier set in the `PARTITION BY` clause. The data sample on the left (Figure 4.3 (a)) considered qualifier *A* as the partitioning key, while the data sample on the right (Figure 4.3(b)) considered qualifier *B* as the partitioning key. As the qualifier chosen for the `PARITION BY` clause changes, the arrangement of data also changes. The most efficient execution in this environment, as covered in the next subsection, implies that each logical partition undergoes the ordering cycle only once. Thus, all members of each logical partition (as imposed by query construction) must be processed in the same instance or location.

4.2.2 Ordering

The sorting stage is defined through the `ORDER BY` clause. Considering the previous established logical partitions, the intra-partition ordering will execute according to the qualifier or expression considered as argument for this clause. The ordering stage takes special relevance for a group of non-cumulative functions such as the ones belonging to the `ranking` class. Briefly, this class of functions always requires a sorting criterion in order to produce a deterministic result. With an empty ordering clause, the result will only be possible for the group of functions belonging to the `cumulative` class (*i.e.,* sum), as they are not order dependent.

The overall cost of the ordering stage varies depending on the underlying chosen architecture for storing and computing the results, but also from the analytical function chosen. In the following expressions, we consider a partition to be the result of the `PARTITION BY` clause and a shard to be the result of the distribution in the underlying data store. R denotes a relational table and R_i one of its ith shards, while P denotes one logical partition and P_i one of its ith portions. $Cost(R)$ and $cost(P_i, R)$ respectively denote the total cost of ordering relation R and the cost of reordering a given partition within the relation.

On the one hand, a common legacy-type architecture benefits from a shared-everything storage where the sorting cost is bounded by the number of tuples to be ordered in each partition, as depicted in (4.1).

$$Cost(R) = \sum_{i=1}^{partitions} cost(P_i, R) \tag{4.1}$$

That is, without data distribution, the elements of a given logical partition are not spread across several locations. Therefore, the sorting cost is only bounded by the number of tuples to be sorted.

On the other hand, the shared-nothing architecture that is commonly employed in a distributed layout incurs additional sorting costs in each of the distributed instances, as depicted in (4.2). That is, in a distributed layout, the total sorting cost is computed from sorting each distinct logical partition in each node.

$$Cost(R) = \sum_{j=1}^{nodes} \sum_{i=1}^{partitions} cost(P_i, R_j) \tag{4.2}$$

However, this only applies if each logical partition is restricted to a single node, defining the locality requirement for optimal execution, which enables the sorting procedure to benefit from the inter-partition independence.

This is usually not the case. Since data is horizontally partitioned, no single node has the complete logical partition in order to satisfy the sorting required in just one cycle. In a distributed configuration, the final node replying to a client is required to re-sort the results, introducing the added sorting cost as depicted in (4.3).

$$Cost(R) = \left(\sum_{j=1}^{nodes} \sum_{i=1}^{partitions} cost(P_i, R_j) \right) + \sum_{i=1}^{partitions} cost(P_i, R) \tag{4.3}$$

4.2.3 Framing

The framing stage introduces a set of possible clauses bounding the rows that build the window (frame) around the current row within a given partition. The frame is available from two separate constructions, the `ROWS BETWEEN` and the `RANGE BETWEEN`. Each one is complemented by formalizing the start and end of the frame, specifying how many units before and after the current row should be considered. In the `ROWS BETWEEN` mode, the row should be considered as the unit. For the `RANGE BETWEEN` mode, the numerical value in the cell should be considered as the unit. These constructions are complemented with boundary constraints.

In the following, we present a brief description of each clause:

- n `PRECEEDING`: The frame starts n units before the current row;
- n `FOLLOWING`: The frame ends n units after the current row.;
- `UNBOUNDED PRECEEDING`: The frame starts at the beginning of the partition;
- `UNBOUNDED FOLLOWING`: The frame ends at the end of the partition.

In contrast to the ordering stage, the framing stage is independent of the aggregate function.

The framing stage builds on the provided ordering, taking into account the current row being considered to introduce the concept of window or frame. The frame is built from a group of adjacent rows surrounding the current row and changes as the current row moves toward the end of the partition. The framing is set by either the `ROWS BETWEEN` or the `RANGE BETWEEN` clauses. The former considers n rows before and after the current row, while the latter restricts the window by creating a range of admissible values and, the current row is considered if the stored values fit in the provided range.[1]

4.3 Cumulative and Ranking Analytical Functions

Modern analytical engines provide a vast array of analytical functions that can be categorized according to their behavior in a simple two class taxonomy: `cumulative` and `ranking`. The former class produces cumulative and moving aggregates and, tipically, they are not order-dependent. The $sum(x)$, $avg(x)$ or $count(x)$ are just some examples of this class of functions. The latter class produces rankings or percentiles and they are typically order-dependent. The $rank()$, $dense\_rank()$ or $ntile()$ are just some examples of ranking analytical functions.

Figure 4.6(a) depicts the result of computing a WF structured as `select analytical_function() OVER (Partition By A Order By D) from table`, immediately before applying the analytical function. The source relation was partitioned according to qualifier `A` and ordered according to qualifier `D`. The following two examples depict the differences between a cumulative and a ranking analytical function. Figure 4.6(b) considers the $sum(D)$ as the analytical function to be computed. The result qualifier is computed by adding all the values of qualifier `D` in each logical partition (*e.g.,* partition 2), and applying that result to all the rows of that partition. Figure 4.6(c) considers the $rank()$ as the analytical function to be computed. The result is bounded by the ordering offered by qualifier `D`, in each logical partition, producing a different result for each row of the partition.

Cumulative analytical functions such as $sum(x)$ are not order-dependent as they are commutative and associative. Such properties largely improve their par-

[1]Typically, the use of this clause is restricted to numeric types.

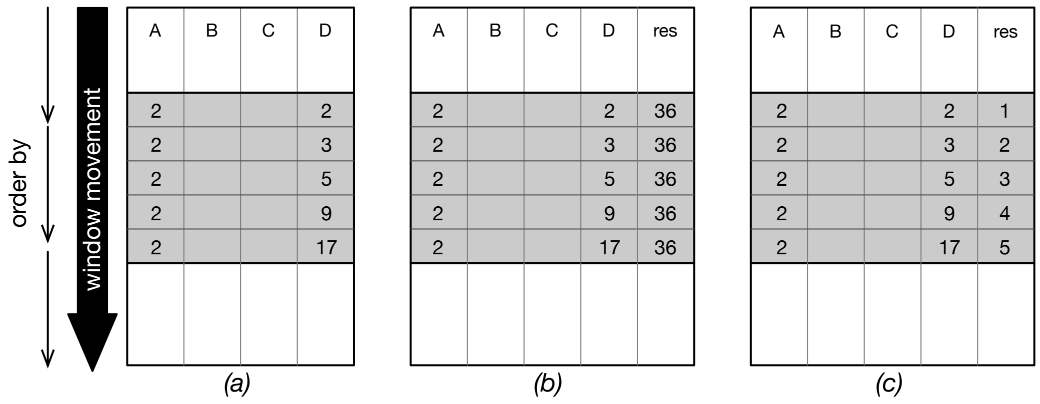


Figure 4.6: WF query as: `select analytical_function() OVER (PARTITION BY A ORDER BY D) FROM table`. (a) WF where the partition by clause generated 1 partitions. (b) Cumulative (sum) analytical function over WF in (a). (c) Ranking (rank) analytical function over WF in (a).

allelization potential. While the commutative property ensures that the absence of an ordering criterion does not compromise a deterministic result, the associative property ensures that parallelization approaches requiring data partitioning satisfy a deterministic result.

Ranking analytical functions such as *rank*() do not benefit from the previous properties since they establish an ordinal association between the rank output and the ordinal variables. This introduces several difficulties in parallelizing the execution when dealing with data partitioning. This is due to a given logical partition being possibly scattered among a group of instances that may not match the physical partitioning in the database, not fulfilling the locality requirement for optimal execution.

On the other hand, the parallelization of cumulative analytical functions is much simpler since the results can be computed in each physical partition without compromising the determinism of the overall final result. Despite the limitation of the latter class of analytical functions, provided that the WF holds a partitioning clause, the parallelization of each logical partition can be achieved as each one can be assigned to a different computing node. Nevertheless, intra-logical-partition parallelism of the ranking analytical functions can be further improved in a distributed setup.

4.4 Distributed Execution

Distributed databases take advantage of data distribution, considering several computing nodes in order to scale query execution. Each database node is split in two layers, the Distributed Query Engine (DQE) and the Data Substrate layer. The latter holds the Data Nodes and manages the data partitions manipulated in each Query Engine (QE) instance.

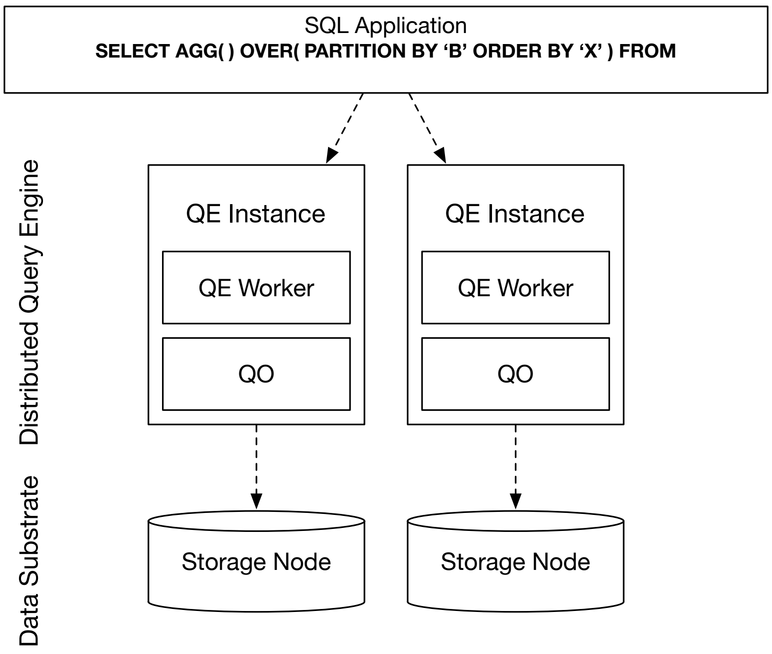


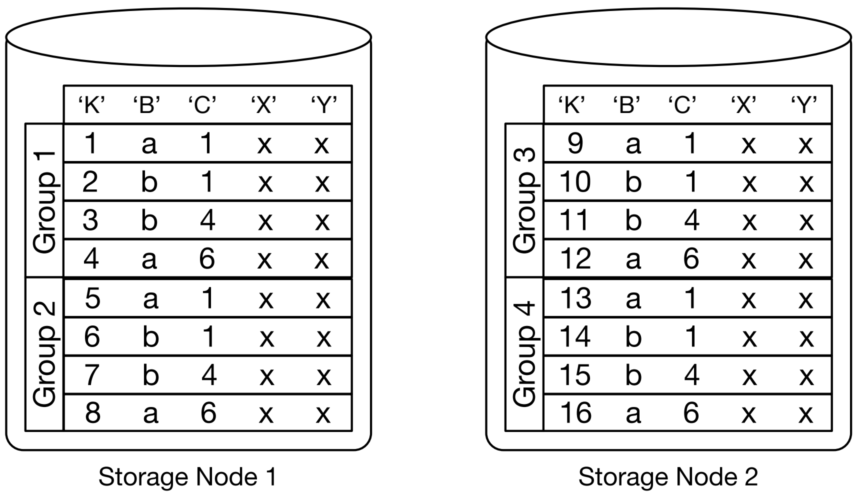
Figure 4.7: Simplified architecture of a distributed query engine.

Figure 4.7 depicts a simplified version for the architecture that models the considered system. Each DQE instance holds a QE, a Query Optimizer (QO) and abstracts a larger set of essential services that are required to cope with distribution, such as the transactional managers in order to ensure data consistency across instances or activity loggers for fault tolerance purposes. The QE translates SQL syntax into sets of single operators, while the QO considers several statistical techniques to improve the execution plan of a query.

In a nutshell, the QE splits the execution of a query into two separate stages: the query planning and the query execution (detailed in section 5.3). During the first stage, the QE decides how the query is executed during the second stage, and which operators are used in such a query plan. The QO uses hints about data in the form of statistical approximations, allowing the QE to optimize query execution based on the approximation cost of each individual operator in a given data set.

4.4.1 Data Splitting

The DQE applies data partitioning techniques in order to distribute data among instances, defining the number of available computing nodes and configuring the installed Degree Of Parallelism (DOP). The data distribution techniques are commonly accomplished by means of a hash function or by assigning ranges of keys to specific storage nodes, considering a single or a collection of attributes as key to guide the splitting decision.

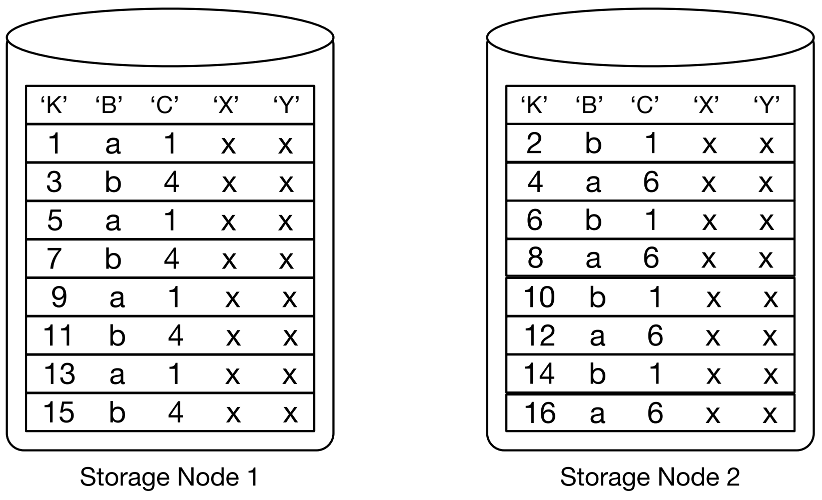


	‘K’	‘B’	‘C’	‘X’	‘Y’
Group 1	1	a	1	x	x
	2	b	1	x	x
	3	b	4	x	x
	4	a	6	x	x
Group 2	5	a	1	x	x
	6	b	1	x	x
	7	b	4	x	x
	8	a	6	x	x

	‘K’	‘B’	‘C’	‘X’	‘Y’
Group 3	9	a	1	x	x
	10	b	1	x	x
	11	b	4	x	x
	12	a	6	x	x
Group 4	13	a	1	x	x
	14	b	1	x	x
	15	b	4	x	x
	16	a	6	x	x

Figure 4.8: Distributed data layout considering K as the partitioning key.

Figure 4.8 depicts the resulting distributed data layout from applying a data splitting strategy across two storage nodes. In this example, data rows are placed in groups composed from ranges of values (guided by qualifier `K`). The number of groups grows as the number of rows in each one exceeds a configurable group size. This distribution strategy is currently considered in a popular data store, HBase [George, 2011].



'K'	'B'	'C'	'X'	'Y'
1	a	1	x	x
3	b	4	x	x
5	a	1	x	x
7	b	4	x	x
9	a	1	x	x
11	b	4	x	x
13	a	1	x	x
15	b	4	x	x

'K'	'B'	'C'	'X'	'Y'
2	b	1	x	x
4	a	6	x	x
6	b	1	x	x
8	a	6	x	x
10	b	1	x	x
12	a	6	x	x
14	b	1	x	x
16	a	6	x	x

Figure 4.9: Distributed data layout considering hash functions as the partitioning key.

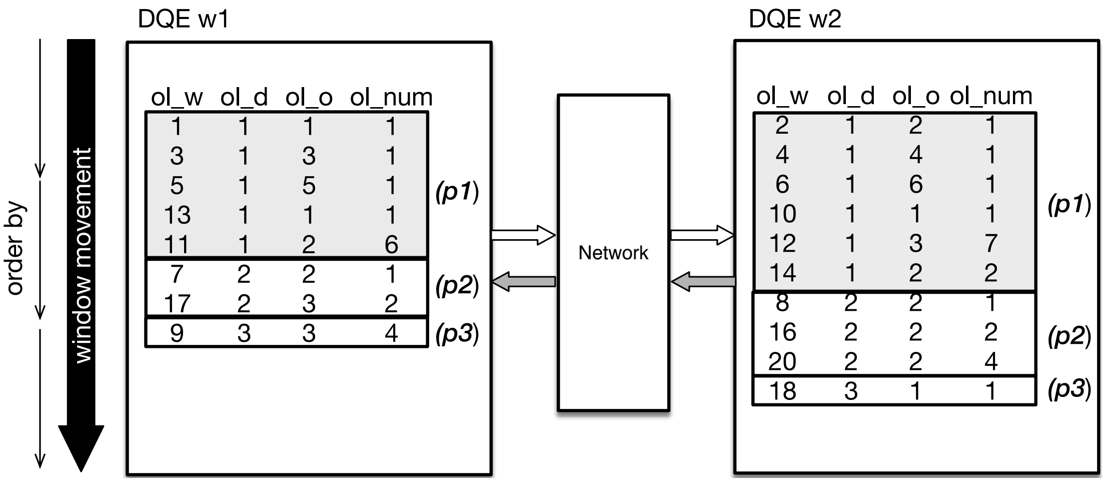
Figure 4.9 depicts the resulting distributed data layout from applying hash functions as the partitioning strategy. Considering hash functions as the underlying data partitioning mechanism, typically induces uniform data distributions, evenly placing sets of tuples across all available storage nodes. This distribution strategy is often designated as *Consistent Hashing* [Karger et al., 1997] and maybe found in data stores such as Cassandra [Lakshman and Malik, 2010].

Despite the partitioning strategy, the data distribution will only favor the attribute (or composition of attributes) as key. In a dynamic query execution scenario, queries may consult several of the available qualifiers, disabling any chance to settle on a specific data partitioning strategy that would favor all qualifiers. A closer look at both figures 4.8 or 4.9 shows that if a query considers, for example qualifier `B` for the `PARTITION BY` clause, no single node holds all the items for that particular logical partitioning scheme (partitions *a* and *b* according to the example). In order to promote intra-partition parallelism, all data rows that build a logical partition may need to be relocated to a remote instance to achieve the co-location of all elements of a logical partition.

4.4.2 Data Forwarding

Data movement during the distributed execution of a WF query is required, ensuring that all the elements of each logical partition are in the same location, in order to fulfill the co-location requirement for efficient execution. In order to forward data while at the same time minimizing the transfer cost, we introduce a statistical technique coupled with a data transfer mechanism, which we designate as shuffler, promoting the co-location of logical partitions. This was achieved by considering a histogram characterizing the universe of elements present in each logical partition, mapping how data is distributed in the storage nodes.

Briefly, the histogram considers the cardinality of each distinct element in each possible column qualifier. Each storage node be characterized by the specific data distribution of the partitioning qualifiers. The introduction of this mechanism along with the shuffler allows data to be forwarded during query execution time to the specific node that, according to this mechanism, is responsible to process a given logical partition.



DQE w1

ol_w	ol_d	ol_o	ol_num
1	1	1	1
3	1	3	1
5	1	5	1
13	1	1	1
11	1	2	6
7	2	2	1
17	2	3	2
9	3	3	4

DQE w2

ol_w	ol_d	ol_o	ol_num
2	1	2	1
4	1	4	1
6	1	6	1
10	1	1	1
12	1	3	7
14	1	2	2
8	2	2	1
16	2	2	2
20	2	2	4
18	3	1	1

Figure 4.10: Shuffling instances partitioned by *ol_w*. In WF context, they were partitioned by *ol_d* and Ordered by *ol_num*. The DQE instances will use the network to combine partitions during execution time. Instance w_1 will hold partitions $ol\_d = 1$, instance w_2 will hold partitions $ol\_d = 2$ and $ol\_d = 3$, respectively.

Consider Figure 4.10 where a table similar to the table in Figure 4.6(a) was split into two partitions in the storage layer. This initial partitioning was defined by hashing the value of the nodes `ol_w` qualifier and performing the arithmetic modulo between the hash result and the number of computing instances ($Hash(value\ in\ ol\_w)\ \%\ \#Nodes$). Guided by the query in Figure 4.1, the results were then ordered according to the qualifier `ol_d` (set as the partitioning clause). Both nodes of the storage layer hold elements from the available three partitions in `ol_d` ($p1, p2, p3$). To promote intra-logical-partition co-location, `ol_d` partitions ($p2$) and ($p3$) in instance $DQE\ w1$ will be relocated to instance $DQE\ w2$ and `ol_d` partition ($p1$) will be relocated from instance $DQE\ w2$ to instance $DQE\ w1$.

4.5 Holistic Shuffling

Indexes [Garcia-Molina, 2008] or Histograms [Poosala et al., 1999] are commonly used by query optimizers as they provide a fairly accurate estimate on the data distribution. This is crucial for a query planner, allowing to map keys to their observed frequencies. Database systems use these structures to measure the cardinality of key ranges. Without histograms, the query planner would need to assume a uniform distribution of data, leading to incorrect partitioning. Consider the following example where a relation R characterizes citizens of three different countries: USA, Germany and Brazil, where each row represents a single citizen. Consider also that USA has 322 million citizens, Germany has 81 million and Brazil has 205 million, thus the relation R would hold a total of 608 million rows. If a given query planner had to deal with a query that would require to partition the data by country into 3 workers, without the histogram it would consider them to be equally distributed, which for this case is not true. This simple example demonstrates the special relevance of relying in histograms when data is skewed [Poosala et al., 1996], a common characteristic of non-synthetic data.

When a query engine has to generate parallel WF query execution plans, each worker would ideally hold a whole logical data partition. However, the cardinality and location of logical partitioning candidates alone do not completely present a heuristic that could be used to enhance how parallel workers would share preliminary and final results. In order to expedite data movement through the improvement of bandwidth usage, the histogram also needs to reflect the volume of data existing in each node, thus creating samples of the size of each candidate logical partition. Together, these metrics allow to co-locate distributed elements of logical partitions while promoting the lowest possible usage of bandwidth in the network interconnect.

4.5.1 Histogram Construction

The histogram considers the row cardinality and average row size for each logical partition. Both could be seen as global metrics that a given query engine may be able to produce and maintain, as this type of information is already used for similar purposes.

The histogram considers the cardinality of each value in each attribute of the relation. Since the construction of the histogram should not be done during query planning time, the QE cannot know beforehand the partitioning clauses induced by queries. As such, we consider all distinct groups of values in each attribute. Each partition will contribute to the histogram with the same number of attributes as the original relation, plus a key, reflecting the data in that partition.

Algorithm 2 Histogram Construction in Partition n

```
1: Initially:
2: $P_n \leftarrow [attr_1, attr_2, attr_n]$
3: $hist\_P_n \leftarrow [key, attr_1, attr_2, attr_n]$
4: procedure COUNT_DISTINCT_KEYS($attr$)
5:     for each $key \in attr$ do
6:         $count \leftarrow number\ of\ key\ in\ attr$
7:         $size \leftarrow size(key)$
8:         $hist\_P_n(key, attr) \leftarrow (count, size)$
9:     end for
10: end procedure
11: function GLOBAL HISTOGRAM($P_n$)
12:     for each $attr \in P_n$ do
13:         COUNT_DISTINCT_KEYS($attr$)
14:     end for
15: end function
```

Algorithm 2 depicts how each partition histogram ($hist\_P_n$) should be built. The selected attribute ($attr$) is traversed and, for each key, the total number of distinct occurrences is computed along with its size. The pair of values is then added to the histogram.

Afterwards, workers share their completed local histograms with a designated master worker, so that the global histogram is assembled. The global histogram considers each physical partition histogram and evaluates, for each key, which is the physical partition that holds the largest volume (in size, evaluating the

$cardinality \times average\_row\_size$). The global histogram is built with the same number of qualifiers from each partition histogram.

4.5.2 Shuffler Action

The holistic shuffler mechanism leverages the data collected by the global histogram to expedite shuffling operations. During the workflow for processing a window operator, there are two different steps where data needs to be shuffled. The first is when the operator starts to reunite logical partitions, thus fulfilling the co-locality requirement. The second is at the end of the operator and reconciles partial results to produce the final result. Therefore, two shuffle operators were considered, each one with a distinct policy: the local shuffle and the global shuffle.

Algorithm 3 Local Shuffle Operation

```
1: Initially:
2: $worker\_id \leftarrow worker\ unique\ identifier$
3: $row \leftarrow [key, attr_1, attr_n]$
4: $hist\_P_n \leftarrow [key, row]$
5: $partition\_by \leftarrow attr_1$
6: function LSHUFFLE($local\_partition$)
7:     for each $row \in local\_partition$ do
8:         $partition \leftarrow row[attr_1]$
9:         $destination \leftarrow hist\_p_n[partition, attr_1]$
10:
11:        if $worker\_id \neq destination$ then
12:            $send(destination, row)$
13:        end if
14:    end for
15: end function
```

The local shuffle shown in Algorithm 3 dispatches rows of a given logical partition to the worker responsible for that logical partition as dictated by the global histogram. As each row is read from scanning the partition, the value contained in that row for the attribute that dictates the partition clause is collected (*partition*). This value is then used together with the partitioning attribute to obtain the destination worker from the global histogram.

The global shuffle shown in Algorithm 4 forwards all aggregated rows to the master worker holding the overall largest data volume. The input data considered

Algorithm 4 Global Shuffle Operation

```
 1: Initially:
 2: $worker\_id \leftarrow worker\ unique\ identifier$
 3: $master\_worker \leftarrow hist\_P_n$
 4: function GShuffle($aggregated\_data$)
 5:     for each $row \in aggregated\_data$ do
 6:
 7:         if $worker\_id \neq master\_worker$ then
 8:             $send(master\_worker, row)$
 9:         end if
10:     end for
11: end function
```

by the global shuffler is composed by the ordered and aggregated rows, both produced during execution time. Such rows will now have to be sent in the master node, whose identity is retrieved from the histogram. Afterwards, as each aggregated row is handled by the operator, it is forwarded to the master worker, if it is not the current one.

Even though the histogram mechanism identifies the ideal worker to compute a given logical partition, forwarding a single row does not maximize bandwidth usage.

Due to the asynchronous nature of the considered DQE, latency is usually not the bottleneck and thus, data movement may be delayed until network usage can be maximized [Gonçalves et al., 2016]. This enables the use of batching in order to improve network usage. A batch payload is formed by grouping rows that need to be forwarded to a common destination and it is regulated by a buffer within the shuffling mechanism, whose size and delivery timeout are configurable.

The understanding of up to what level a given logical partition may or not benefit from batching is shaped in a correlation mechanism, regulating the decision by identifying the logical partitions that are good candidates to form batches.

4.6 Similarity Awareness

The Query Optimizer (QO) considers several statistical mechanisms to explore data features, in order to improve query execution performance. Without them, independence assumptions between attributes are preserved, which commonly

Algorithm 5 Similarity Aware Shuffling Mechanism

```
 1: Initially:
 2: $P(r) =< r_0, r_1, r_2, r_n > \leftarrow partition$
 3: $r_i \leftarrow current\_row$
 4: $pbk \leftarrow partition\_by\_key$
 5: $w\_id \leftarrow worker\_id$
 6: $H \leftarrow histogram$
 7: $t \leftarrow similarity\_threshold$
 8: procedure Similarity($attr\_A, attr\_B$)
 9:     $Sim \leftarrow \frac{unique(attr\_A \cap attr\_B)}{unique(attr\_A \cup attr\_B)}$
10: end procedure
11: procedure BatchShuffling($P(r), dest$)
12:     $send\ P(r)$ to $dest$
13: end procedure
14: procedure HashShuffling($r_i, dest$)
15:     $send\ r_i$ to $dest$
16: end procedure
17: function Shuffler
18:     $dest \leftarrow H(r_i.pbk)$
19:     if $w\_id \neq dest$ then
20:         $Sim \leftarrow$ similarity($w\_id\_pbk$,$dest\_pbk$)
21:         if $Sim > t$ then
22:             BatchShuffling($P(r), dest$)
23:         else
24:             HashShuffling($r_i, dest$)
25:         end if
26:     end if
27: end function
```

leads to under or over provisioned query plans. As in real-world data, correlations between relation attributes are the rule and not the exception, the array of correlation or other algebraic extraction mechanisms in the literature is vast, namely [Brown and Hass, 2003], [Fan et al., 2008] and [Liu et al., 2016]. Correlations can also be used in DQEs to improve how data distribution is handled. When logical data partitions need to be relocated to improve co-locality, the correlation between qualifiers in different locations of the storage layer can be explored to minimize the required data movement.

The *similarity measure* quantifies to what level the logical partitions of a given attribute held by different storage nodes are alike. Data partitions with high *similarity* are good candidates to be shuffled in a batch payload. This is so as a high *similarity* implies a high common number of logical partitions. On the other hand, data partitions with low *similarity* are better candidates to be immediately shuffled for their destination. This is so as they share a low number of common partitions.

This is efficiently achieved through Algorithm 5. The *similarity measure* quantifies in a universe between 0 (not similar) and 1 (similar) how similar two attributes are, by considering the number of unique values in each one of them to compute the metric. The algorithm is considered during the first shuffling stage. It considers each logical partition ($P(r)$), the previously introduced Histogram (H) and a configurable *similarity* threshold. Three auxiliary procedures are considered. The `SIMILARITY` procedure computes the *similarity measure* from the set of unique values in the qualifiers considered as argument. The `BATCHSHUFFLING` procedure marshals all the rows of partition $P(r)$ and sends it to the destination worker *dest*. The `HASHSHUFFLING` procedure marshals a single row r_i and sends it to destination *dest*.

When the shuffler action is required, it uses histogram H to verify what is the optimal destination from row r_i. When the destination is a remote instance (line 19), the shuffling mechanism computes the *similarity measure* between the local ($attr_A$) and destination ($attr_B$) qualifiers (line 20). The partition $P(r)$ is marshaled to the appointed destination when the observed *similarity* is above threshold t (line 22) (`BATCHSHUFFLING`), or each row r_i is otherwise sent to destination (line 24) (`HASHSHUFFLING`). The parameter t sets a threshold above which rows are forwarded in batch to the destination instance. This parameter defaults to 0.5 meaning that if not modified, rows are batch forwarded if the origin contains at least half the number of unique partition values of the destination.

4.7 Evaluation

Along the current section, we present the evaluation for the Holistic and Similarity mechanisms.

Holistic shuffling

In order to evaluate the Holistic shuffling technique, we considered RX-Java [RXJ, 2015] to simulate the parallel execution of the window operator in several workers. This framework provides bindings to the Java language, enabling it to use the semantics of Reactive Programming [RX, 2015]. We selected this framework as it allows to establish a series of data streams, mimicking the window operator data flow. Throughout the evaluation, the single ranking query in Figure 4.11 was considered, holding a window function over TPC-C'c synthetically-generated `ORDER_LINE` relation, characterized by 10 distinct column attributes. The values considered for each one of these attributes were distributed according to TPC-C's specification. Generated data is characterized by 100 distinct logical partitions, each one with 500 rows. Globally, the configured relation held 50×10^3 tuples.

```
select rank() OVER( PARTITION BY OL_D_ID ORDER BY OL_NUMBER) from
                              ORDER_LINE
```

Figure 4.11: Test WF query for the holistic technique.

Experiments were performed on a system with an Intel i3-2100-3.1GHz 64 bit processor with 2 physical cores (4 virtual), 8GB of RAM memory and SATA II (3.0Gbit/s) hard drives, running Ubuntu 14.04 LTS as the operating system.

For comparison purposes, we report the results by using a naive approach and the Holistic Shuffler. The naive approach does not consider any knowledge to forward data, broadcasting data among all workers.

The results in Figure 4.12 and Figure 4.13 are presented in a logarithmic scale, as the average of 5 independent runs for each configuration.

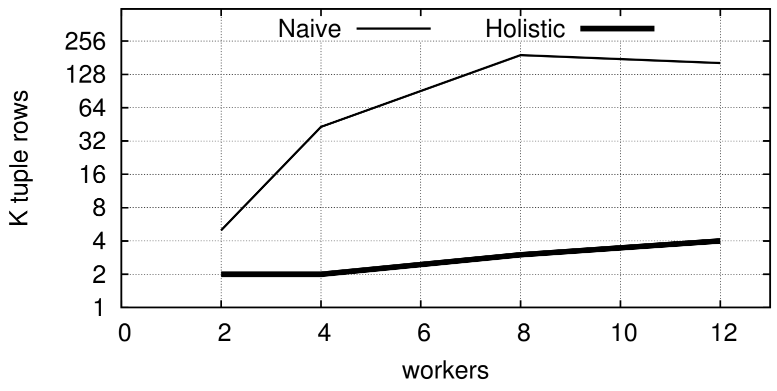


Figure 4.12: Average forwarded rows for the local and global shuffling stage.

In Figure 4.12, the Shuffler technique required on average only 14.7% of the rows required for the naive approach to reunite all the logical partitions in each computing node. The results regarding the Local Shuffling stage are depicted in Figure 4.13. This experiment varied the number of computing nodes that participate in the computation of the ranking query, verifying the number of rows that were forwarded according to each technique.

The large difference is justified by the fact that the naive approach reunites logical partitions by forwarding data among all participating nodes, which intrinsically creates duplicates in each node.

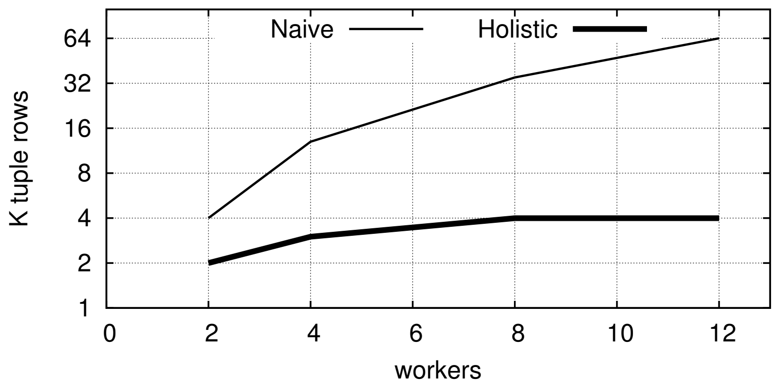


Figure 4.13: Forwarded rows for the local shuffling stage.

Algorithm Phases

The complexity distribution for each stage of the WF environment is depicted in table 4.1. It provides a detailed description regarding the execution time for configurations with 2 and 8 workers. It shows that considering the Holistic Shuffler provides better results in each step of the computation, reducing the number of shuffled rows in each shuffling stage. As a consequence, the ordering and ranking operations also benefit since each node holds less rows to be processed, reducing the overall computing time.

	2 workers		8 workers	
stage	Naive	Holistic	Naive	Holistic
Local shuffle [K rows]	499	200	3596	399
Sort [ms]	537	252	201	15
Rank [ms]	20	18	26	8
Global shuffle [K rows]	591	299	34763	1000

Table 4.1: Performance of each step for 2 and 8 workers for the ranking WF computation.

Similarity

We validated that by batch shuffling tuples between DQE instances we save bandwidth, improving execution time of the shuffling stage. We considered a synthetic data set and shuffled rows between distinct DQE instances. The data set used was extracted from the TPC-DS [Transaction Processing Performance Council, 2012]. We extracted a single relation (`WEB_SALES`) which is composed of 35 distinct attributes, configuring TPC-DS with a scale factor of 50GB. This resulted in a relation with 9.4GB corresponding to 36×10^6 rows.

The outcome of the mechanism we propose is directly related with the data distribution considered. In order to bound the outcome in terms of the lower and upper performance values, we statistically analyzed the considered relation. The lower bound is set by not using the *similarity* mechanism. The upper bound is set by considering the relation attributes that would favor data distribution. This was achieved by identifying the placement key attribute, but also a candidate attribute to be the partitioning clause or shuffling key (Partition By Key (PBK)) of the WF. The placement key attribute will define the data distribution in each DQE Storage Node through the use of a Hash function, and the PBK will define the runtime partitioning within the WF environment.

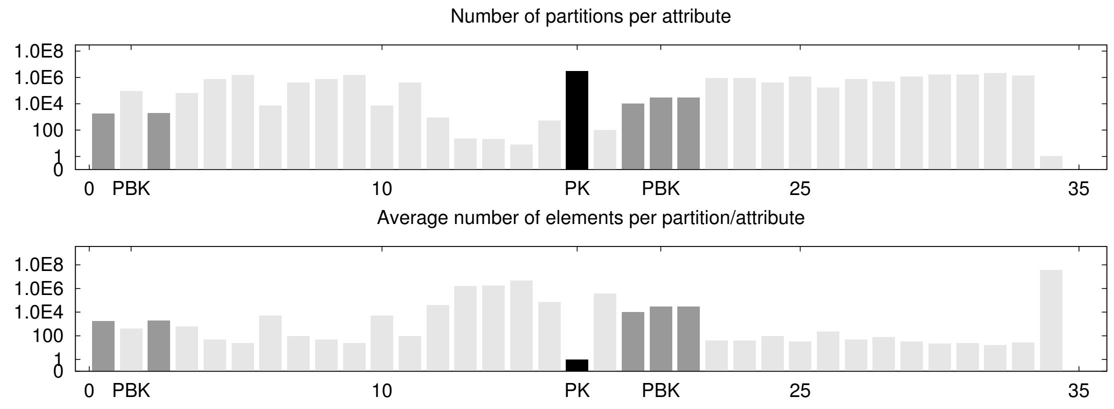


Figure 4.14: Number of partitions per attribute (top) and the average number of elements per partition/attribute (bottom).

The results are depicted in Figure 4.14. The top plot presents the number of partitions for each single attribute in the considered relation. That is, the number of unique values in each attribute. The bottom plot depicts the average cardinality of each partition. That is, the average number of elements in each group of unique values in each one of them. The horizontal axis represents the attribute index, while the vertical axis quantifies each measure in logarithmic scale. The attribute considered for placement key (PK) is shown in black and the candidates for WF Partition By key (PBK) are shown in dark gray.

The ideal candidate attribute to become the relation placement key is the attribute that displays the highest partition number and at the same time holds the smallest cardinality, ensuring an even data distribution and reduced data skew. On the one hand, observing both plots leads us to consider attribute with index 17 (`ws_order_number`), displaying the highest number of partitions, each one with a single element. On the other hand, the candidate attributes to be selected as WF PBK are the attributes that would hold at the same time a high number of partitions and high partition cardinality. These are good PBK candidates since they will induce a number of logical partitions that is above the configured DOP. The observation of the plots leads to identify as candidates the attribute indexes depicted in dark gray, from which we select attribute 0 (`ws_sold_date_sk`) as PBK.

After the election for the `PK` and PBK keys, we conducted a second experiment to verify the computed *similarity measure*. Figure 4.15 depicts the results of applying the metric in two scenarios. In both cases, we consider our scenario to be built from several DQE instances and corresponding Storage Nodes.

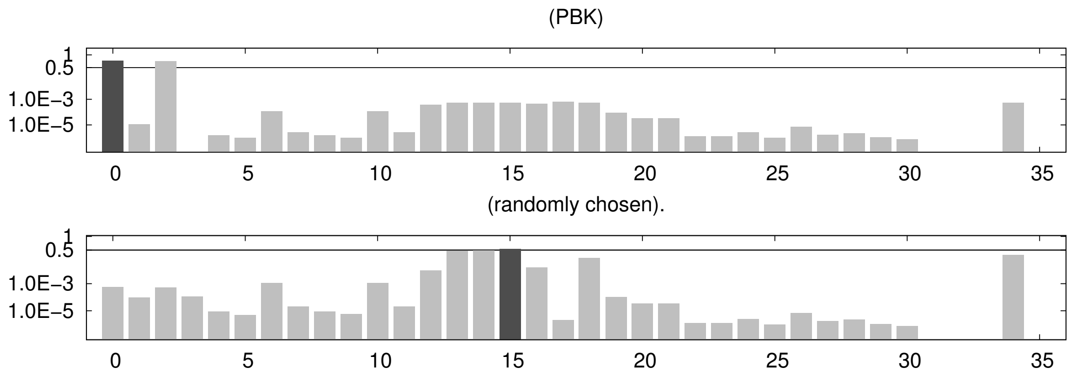


Figure 4.15: Similarity between qualifiers in two data nodes. Horizontal axis represents the attribute index. Vertical axis represents the *Similarity measure* in logarithmic scale.

On all experiments, we considered only the communication layer of the DQE where our contribution is, thus avoiding the SQL parsing and optimization stages. Each data partition was computed by applying a Hash function with the elected `PK` dividing the data into as many partitions as configured DQE instances. We first considered the configuration with 2 instances A and B. In the experiment in the top plot we computed the *similarity measure* between the the PBK of location A and each distinct attribute in location B. It is possible to observe that attribute 0 in location B presents the highest similarity, followed by attribute 2. These are also the only attributes that are above the set up threshold of 0.5 denoted by the horizontal line. The remaining attributes have a residual similarity measure. The bottom plot depicts a different configuration where attribute 15 was randomly chosen among all non-candidate attributes. The similarity measure in this attribute is lower than our threshold, even though it seems to be equal given the logarithmic scale required to observe the remainder attributes. Therefore, the results achieved during the first configuration would induce the shuffler to use batching mechanisms to forward partitions among DQE instances, instead of hash forwarding. The latter would culminate in sending a single row at a time.

In order to verify the impact regarding network usage, we conducted an experiment to assess the magnitude of the network savings promoted. Namely, we considered configurations with 2, 4 and 8 DQE and Storage instances. The computing nodes were only set up with the communication layer responsible for

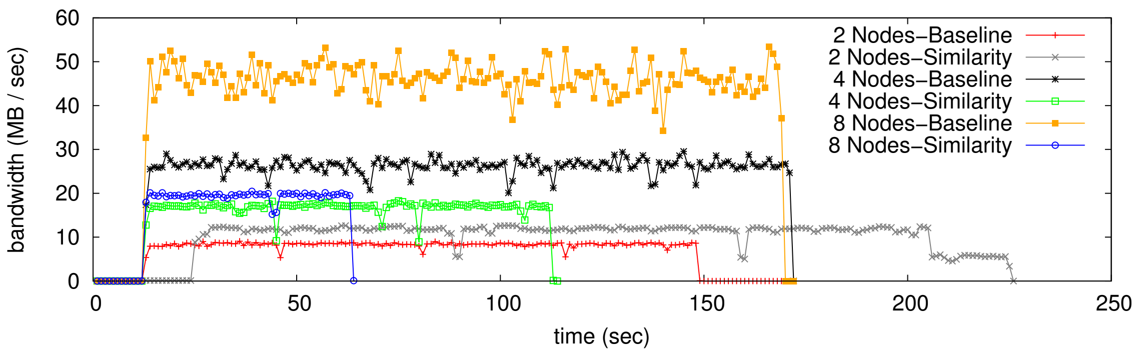


Figure 4.16: Bandwidth (outbound) registered during shuffling between instances.

the shuffling in the WF environment. Each node is comprised of commodity hardware, with an Intel i3-2100-3.1GHz 64 bit CPU with 2 physical cores (4 virtual), 8GB of RAM memory and one SATA II (3.0Gbit/s) hard drive, running Ubuntu 14.04 LTS as the operating system and interconnected by a switched Gigabit Ethernet network. During execution, each computing node acts as a DQE instance shuffler, forwarding data to the remainder instances. In a distributed deployment, the DQE instance will be co-located with other services (e.g., storage node) which will typically restrict the available memory to the DQE instance.

We evaluated two configurations where the first represents a baseline comparison, forwarding all data by hash shuffling, and a second where data is forwarded according to our *similarity* mechanism.

The results depicted in Figure 4.16 are twofold. The *similarity measure* registered both a decrease in bandwidth and it also promoted a shorter execution period for the shuffling technique. This is the result of pairing the batch shuffling mechanism together with the proposed *similarity measure*. The savings induced come at a residual cost, since the statistical information is not collected for the single purpose of this improvement, nor it has to be updated in each query execution. The *similarity measure* technique only proved effective from the configuration with 4 instances onward, since it is only from that configuration that both bandwidth and execution time are lower than the baseline. For the configuration with only two nodes, the baseline technique proved to be better by both shortening the shuffling time and registered bandwidth. However, in the configurations with 4 and 8 nodes, the *similarity measure* was able to reduce the bandwidth and execution time when compared with the baseline approach. As

	2 nodes	4 nodes	8 nodes
Baseline (MB)	1,132.45	4,172.59	7,237.56
similarity (MB)	2,365.34	1,695.24	991.72
Bandwidth Gain (x)	-0.48	2.46	7.30
	Shuffle time		
Baseline (sec)	149	172	170
similarity (sec)	226	114	65
Speed up (x)	-0.48	1.55	2.61

Table 4.2: Total bandwidth (sent) and execution time registered for each configuration.

the number of partitions in the system increase, each single partition becomes responsible for a shorter set of data, promoting bandwidth savings up to 7.30 times for the 8 node configuration.

The previous experiment evaluated the shuffling mechanism by considering an attribute with ideal *similarity measure* and partitioning on the storage layer. In order to demonstrate the impact of selecting an attribute that does not favor a uniform distribution of data among data partitions, we conducted a second experiment that considered an attribute with poor partitioning properties (*i.e.,* reduced number of partitions). The results consider the same component configuration, but selected attribute 15 (`ws_warehouse_sk`) for the partitioning. When selecting an attribute that lacks the desirable distribution, the logical partitions will present an imbalance, thus promoting a low *similarity measure.* Therefore, the shuffling mechanism will not be able to maximize network usage and will end up having to consider the `HASHSHUFFLING` mechanism to forward data. However, we point out that they are in line with the considered baseline results presented in table 4.2, registering a bandwidth variance of $\pm 4\%$. Moreover, even though we do not consider it, the use of compression techniques may further increase the observed savings.

4.8 Related Work

Window functions were introduced in the SQL:2003 ANSI SQL Standard. They provide a powerful, yet simple syntax, that enables complex queries to be expressed. These functions allow different aggregation types, namely: cumulative or ranking, to be computed over a group of logical data partitions. Most leading database systems now offer the ability to interpret WF or at least a sub-set of the capabilities of the environment, namely: Oracle [Corporation, 2015], IBM DB2 [IBM, 2013], Microsoft SQL Server [Coporation, 2013], SAP Hana [SAP, 2014], Cloudera Impala [Kornacker et al., 2015] or PostgreSQL [Postgresql, 2015].

Despite its relevance, parallel implementations and optimizations considering this operator are almost non-existing, to the best of our knowledge, we feel that the few existing works can be categorized using a simple taxonomy: *Complexity Reducers* or *Many-Core Parallelization strategies*. The first category holds works that either try to reduce the complexity of the window operator or use it as a tool to reduce the complexity of queries that originally did not hold any WF. Works such as [Cao et al., 2012] or [Zuzarte et al., 2003] fit in the first category, respectively tackling optimization challenges related with having multiple window functions in the same query, and showing that it is possible to use them as a way to avoid sub-queries and lowering quadratic complexity. However, such approaches do not offer parallel implementations of this operator. Also in the first category, a vast array of correlation mechanisms have been so far deeply studied in the literature. Nonetheless, most of the conducted studies focus on efficient ways to discover and exploit soft and hard correlations [Ilyas et al., 2004], allowing to find different types of functional dependencies.

The second category holds works that try to expedite the execution in architectures composed by servers with multiple CPUs per machine, thus improving parallelization. The distributed architectures powered by the *cloud computing paradigm* must accommodate a third and new class: *Distributed Strategies*, where execution is spread through a group of several computing instances. Works like [Leis et al., 2015] introduced mechanisms to improve the performance of the WF environment when many-core architectures are used. Distinct approaches and algorithm improvements are introduced, enabling to parallelize the distinct stages of the operator. Finally, the third category holds works that consider distributed architectures to improve scalability of the WF operator. This contribution is placed in this final category.

When addressing WFs, a common misconception generally brings a comparison between SQL WF (in which our contribution focuses) and Complex Event Processing (CEP) windowing. Differences are both semantical and syntactical. On the one hand, the CEP environment is characterized by an incoming and infinite stream of events. From there, a configurable, but constant sample (*e.g.,* window) builds a sketch [Garofalakis et al., 2013] where aggregations are derived. On the other hand, SQL WF are computed over finite sets built from SQL relations. While the former windows are fixed and the data moves through, in the latter, the data is fixed and the window performs the movement. Moreover each approach considers distinct SQL keywords (*e.g.,* `OVER`, `RETAIN`) and subsequent syntax.

4.9 Remarks

This chapter focused on a specific class of analytical functions, introducing WF's syntactical interface and covering important aspects regarding their organization, implementation and, most importantly, the key factors that hurdle the way toward the parallel execution of these functions (*e.g.,* holistic sorting of partitioning). The particular challenges associated with the distributed execution of this analytical class were presented. Moreover, two distinct but complementary techniques were introduced in order to mitigate the lack of holistic awareness for the distributed placement of logical partitions.

Results show that considering the Holistic shuffling technique promoted and improvement in the number of rows that effectively had to be moved; requiring to shuffle only 14.7% of rows when compared with the naive approach. Moreover, the *similarity measure* considered how remote partitions of a given column qualifier relate in order to improve data movement through the use of batching mechanisms, rendering a speedup of 2.61 when compared with the baseline approach considered.

Chapter 5

Hybrid Query Engine Integration

This chapter addresses the architecture of the Hybrid Distributed Query Engine (HDQE). Moreover, it addresses how analytical queries are deployed while taking advantage from the parallel execution capabilities. It addresses the impact that data partitioning has on the locality requirement for analytical execution, particularly in the WF environment.

The rest of this chapter is organized as follows: section 5.1 addresses the HDQE architecture. Section 5.2 details the data partitioning mechanism, while section 5.3 presents the parallel query strategies deployed in the HDQE. Section 5.4 evaluates the system and section 5.5 presents related work. Finally, section 5.6 discusses the achievements.

5.1 Architecture

The architecture of the HDQE is based in a highly transactional Platform as a Service (PaaS) [Jimenez-Peris et al., 2015]. The hybrid processing capabilities, means that it can process transactional (OLTP) and analytical (OLAP) workloads over the same dataset. Our contribution focuses on the OLAP part, providing a parallel implementation of WF that leverages the distributed nature of the HDQE. The distributed design merges the horizontal scaling capability offered by NoSQL databases (*e.g.,* HBase [George, 2011], Cassandra [Lakshman and Malik, 2010] or Riak [Klophaus, 2010], etc.) with the consistency guarantees provided by common RDBMS (*e.g.,* Oracle [Corporation, 2015], IBM DB2 [IBM, 2013] or SAP HANA [SAP, 2014], etc).

The HDQE architecture is depicted in Figure 5.1 and builds a multi-tier system comprised of several layers, namely: the Query Engine (QE), Query Optimizer (QO), Transactional Manager (TM) and the Data Substrate layers. The Query Engine offers a standard relational SQL interface and is responsible for planing and executing the queries received by clients through a JDBC interface. The Data Substrate takes the responsibility to store and maintain data persistent. A Zookeeper [Hunt et al., 2010] ensemble is used to provide coordination services to all components. Components are spread along a set of computing instances that are typically, but not necessarily, symmetric.

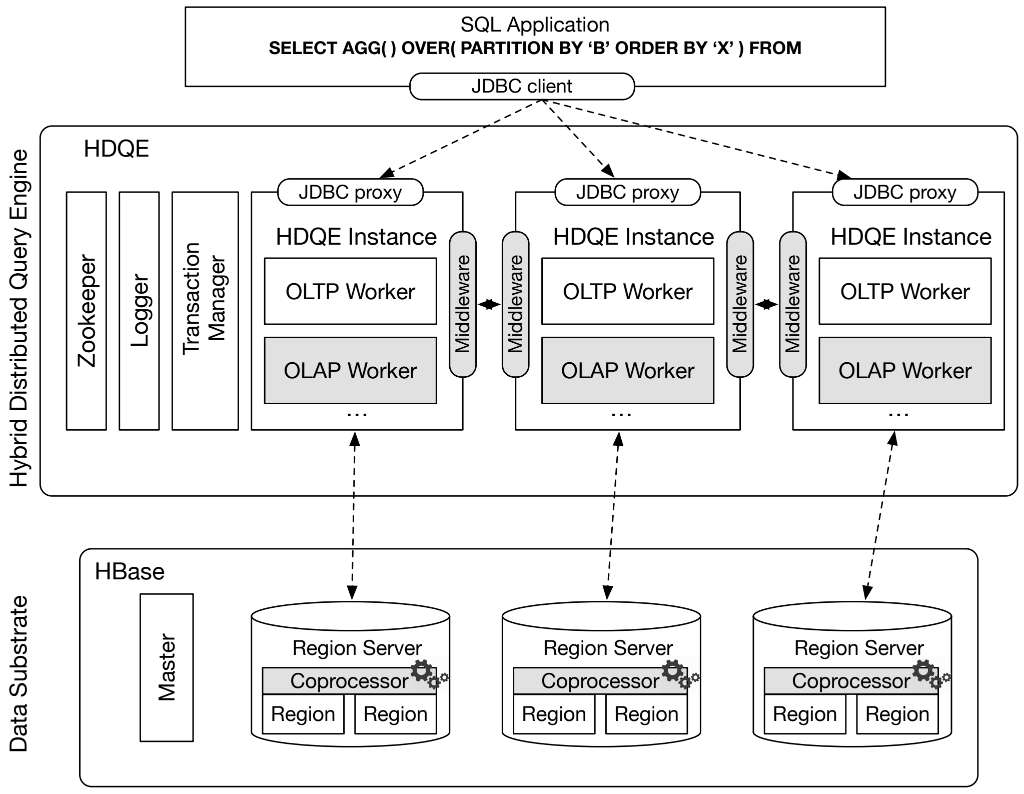


Figure 5.1: Hybrid Distributed Query Engine architecture.

The HDQE is built on the Apache Derby Query Engine [Apache, 2016] (*i.e.*, including the compiler and query runtime), while the Data Substrate is HBase [George, 2011]. HBase is an elastic key-value data store that spreads data among a group of instances named RegionServers. Previous work introduced several modifications to Derby in order to replace its file based storage with HBase as its persistence layer [Vilaça et al., 2013]. This includes the required translation from the relational into the appropriate schema of the data store.

Tables translated from the relational schema are horizontally split into a set of partitions designated as Regions. Each RegionServer is then accountable for a group of Regions where data resides. HBase uses the Hadoop Distributed File System (HDFS) [Shvachko et al., 2010] for replication and fault-tolerance purposes. Moreover it introduces a mechanism designated as CoProcessors that allows execute of custom server-side code. Each HBase RegionServer is typically, but not necessarily co-located with one HDQE instance.

The transaction management provides the ACID properties to all layers. Moreover, the transaction management implementation relies on distinct components, which can be scaled independently, to provide each ACID property. Together with other optimizations, such as asynchronous messaging and batching, it allows the HDQE to scale to hundreds of nodes [Jimenez-Peris et al., 2012].

The previously described components build the core of the system regarding the execution of OLTP operations. Moreover, the query engine component of the HDQE is also able to perform OLAP operations. The OLAP architecture is blended with the previously described components. This way, both the OLTP and OLAP operations share a common data source, bypassing the need to migrate data to a secondary database engine. The visibility for OLAP operations is provided through the Snapshot Isolation criterion [Berenson et al., 1995], bounding the visible tuples to the most recent updates on query start time.

The HDQE was extended with the ability to perform WF, particularly addressing the provisioning of parallel execution capabilities. The contribution presented lies within the components depicted in gray and it is discussed in the following sections.

5.2 Data Partitioning

Data partitioning in the HDQE is provided by HBase's scalable design. This is achieved through the individual management of Regions, distributing them across the available RegionServers. More Regions are created when the configurable Region size is reached or by custom request. In runtime, HBase resorts to a data placement component to reassign Regions to different RegionServers, thus

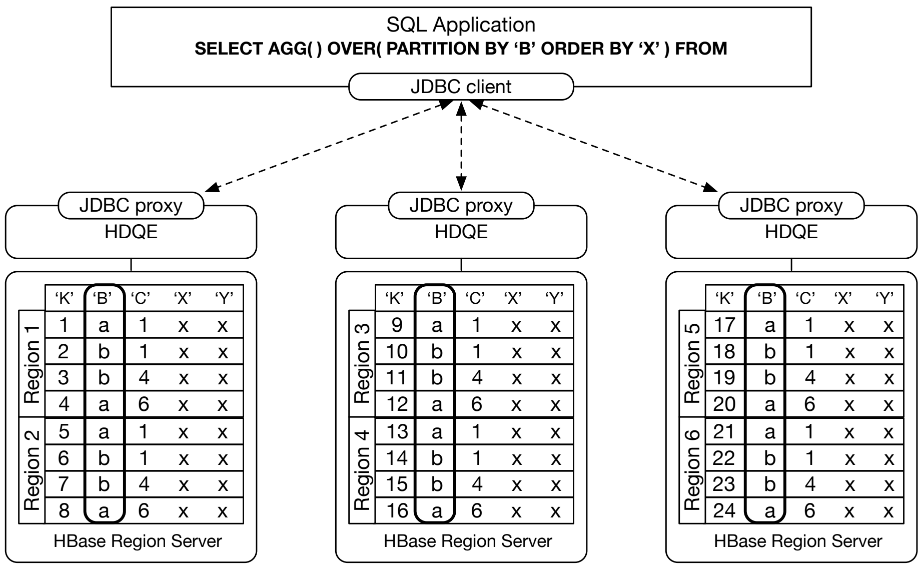


Figure 5.2: Data partitioning strategy.

ensuring an even load throughout the cluster. Each region is composed by sets of rows with primary key values, lexicographically sorted. Each row may have a different number of qualifiers. With row placement governed by the key in each row, several regions may contain a given column qualifier, which can then be spread along several RegionServers.

Figure 5.2 depicts a table composed by a key and 4 column qualifiers. Considering the query in Figure 5.2 and the establishment of logical partitions governed by qualifier B, the horizontal partitioning provided by the data substrate prevents the use of intra-partition parallelism as each logical partition is spread across all nodes. Any attempt to execute the plan for this query in parallel would incur in the extra cost presented in expression 4.3 as data partitioning considers only the table's primary key, hence, there is no co-location of logical partitions. In a dynamic environment characterized by a hybrid execution where queries consult several possible qualifiers, it is not possible to settle on a specific data partitioning strategy that would favor all qualifiers. We therefore consider a mechanism that dynamically forwards data during query execution, favoring the qualifier considered in the query's *Partition by* clause. This way, the elements of each logical partition will share the same location, fulfilling the locality requirement.

5.3 Parallel Query Planing and Execution

The parallel implementation of the HDQE for OLAP queries follows the single program multiple data (SPMD) [Darema, 2001] model, where multiple symmetric workers (threads) on different HDQE instances execute the same query, but each of them deals with different portions of the data.

When a connection is established, the connection thread (the Master worker) uses a coordination middleware that handles the initialization of the multiple worker threads, and for each query received, it broadcasts the query plan to all workers and collects the results. Each query goes through a prepare stage where the query plan is assembled, optimized and converted to bytecode. Next, the execute stage executes the prepared plan, producing the query results.

Query plans need to be adapted so that they can be executed in parallel. The HDQE starts by producing an optimized query plan for single-threaded execution, which is later adapted for parallel execution, as depicted in Figure 5.3. This requires mainly two changes:

- restrict the scan operators that read the table data from the data store, to disjoint ranges of the data;
- rewrite sub-trees of the query plan involving stateful operators, adding the needed shuffle operators.

This enables the same worker thread to process related rows in the same location. The former relies on a scheduler, which divides the table range requested by the query in multiple disjoint shards, and assigns each of them to a different worker.

For the latter, shuffle operators are used to redirect rows to a certain worker based on a key, typically a hash-code (computed from the attributes used as key by the stateful operator being parallelized). Additionally, shuffle operators are used to collect all rows in the Master worker at the end of the query execution (as the Master worker is in charge of returning the result rows to the client), and to broadcast rows from sub-queries.

When the execution starts, the schedules for scans are initialized to determine the shards that each worker will read. The schedule is built based on histograms of the tables, which allows to partition the scan range in multiple shards of similar

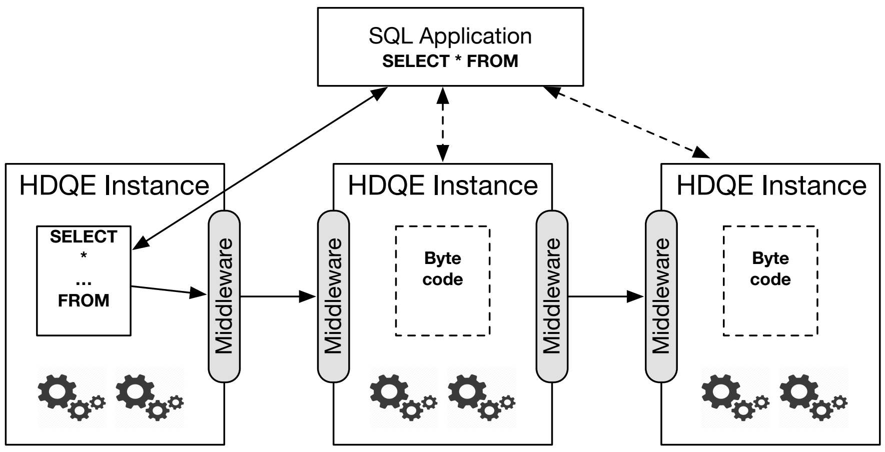


Figure 5.3: Shared bytecode query plans among HDQE instances.

size. Moreover, the histogram contains information regarding the location of each shard. This enables the scheduler to assign shards to workers, and when possible, collocate them with the RegionServer holding the shard being handled, in this way optimizing data locality.

5.3.1 Query Planing

In order to take advantage of the distributed nature of the architecture, the prototype distributes the execution of logical partitions across all available HDQE instances, while considering the parallel execution of WF with non-cumulative order bound aggregations.

Figure 5.4 depicts the distributed logical query execution plan for a WF computing a ranking aggregation, with a partitioning and ordering clause, each one considering a single argument.

The numbers depicted in Figure 5.4 dictate the relative execution order between stages in each participating HDQE instance. Data is first collected through a range-scan in each instance (step 1). At this point data may need to be relocated to a different HDQE instance. This is achieved by consulting a pre-computed histogram, verifying the ideal location (the location that requires to transfer the lowest amount of data) of the value within the qualifier serving the shuffling stage (the qualifier provided in the *Partition by clause*). This is performed in the Shuffle Histogram stage (step 2), that forwards the data row if

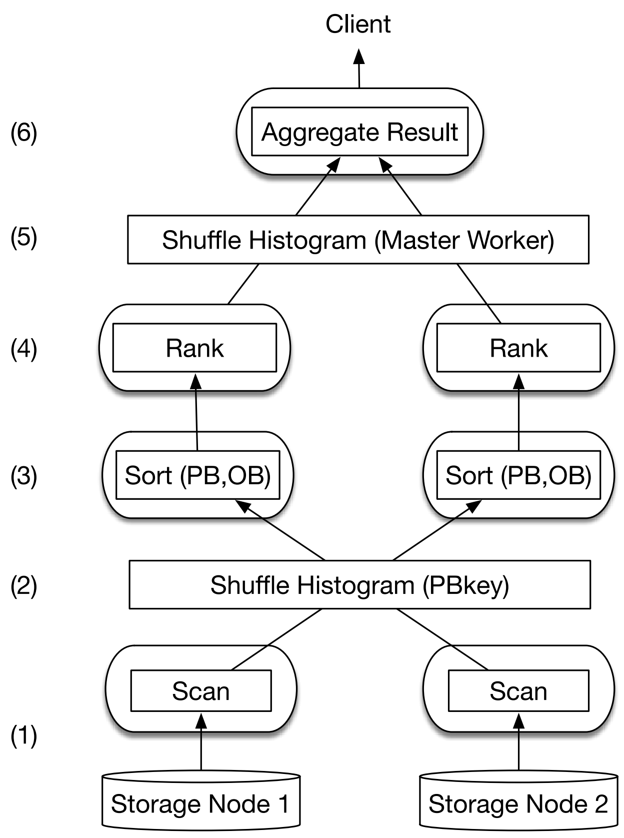


Figure 5.4: Parallel distributed query plan for ranking window function.

the computing location reside in a remote instance. Afterwards, rows arriving from either the local storage or from a remote location are locally sorted. The considered statistical mechanism enables the co-location of rows belonging to the same logical partition in the same HDQE instance. This ensures that all members of each logical partition are located in the same instance and, since logical partitions are independent, the holistic ordering is not compromised. After the sorting stage, the aggregation itself is computed and the partial result is sent to the Master worker to be delivered to the JDBC client that issued the query.

5.3.2 Shuffle Histogram

The shuffling mechanism pre-analyzes the distributed Regions that exist in all RegionServers, to relocate rows to remote HDQE instances. That is, it verifies which is the ideal location for each qualifier to be computed based on the stored values. This was achieved by employing the Shuffler introduced in Section 4.5, providing a mechanism to learn about the location of distributed data partitions, and creating a statistical data sketch that maps the ideal location for distributed data partitions.

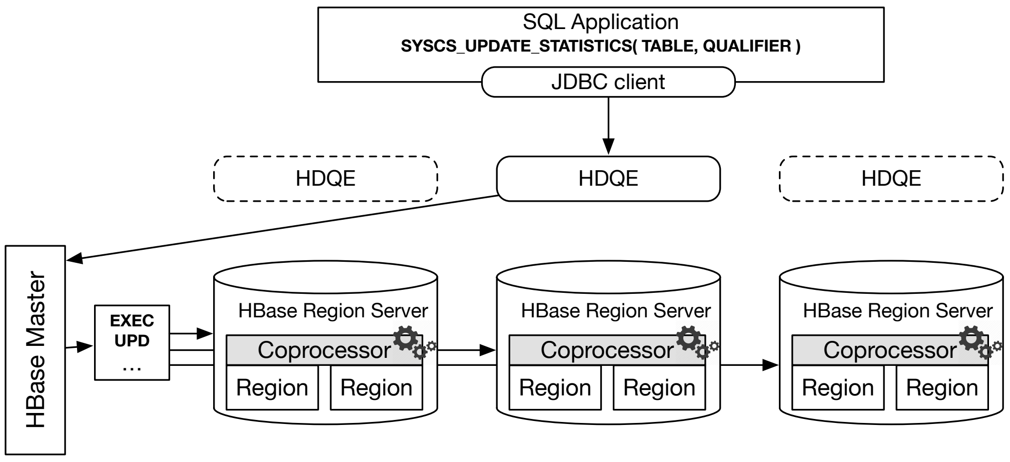


Figure 5.5: Update statistics via HBase CoProcessor.

We built on this contribution and refer to the architecture in Figure 5.1 to place this mechanism as close as possible to data, considering HBase's CoProcessor framework. This way, the impact of computing statistics is minimal, as data transfer to a client computing the statistics outside RegionServers is avoided. This framework establishes several types of CoProcessors to be used under different circumstances, namely: *Observer* or *EndPoint* [George, 2011]. Briefly the first type transparently reacts to client issued operations such as put, read and scan; while the second type reacts to a specific client request. We focus on the *EndPoint* type, as it behaves as a common *stored procedure* in a RDBMS, being triggered by a client Remote Procedure Call (RPC).

The statistics are computed for a given qualifier of a given table by issuing an `UPDATE_STATISTICS` call as depicted in Figure 5.5. Any HDQE instance is able to handle the request which is forwarded to HBase through its Master node. The HBase Master node keeps track about which RegionServer is serving a given Region at any point in time. Likewise, the HBase Master asynchronously forwards the request to the RegionServers that are responsible for the Regions where the specific table is stored.

Algorithm 6 CoProcessor update statistics stage 1

```
 1: Initially:
 2: $q \leftarrow Input\ qualifier$
 3: $Rs\_id \leftarrow RegionServer\ ID$
 4: $Region\_List \leftarrow Region\ ServerRegions$
 5: procedure CREATE_TABLE($RegionServer\ ID, qualifier\ q$)
 6:     $RegionServer \leftarrow add\ table\ with\ column\ RegionServer\ ID$
 7: end procedure
 8: procedure STAGE_1($qualifier\ q$)
 9:     $Map(value, count) \leftarrow null$
10:     for each $region \in Region\_List$ do
11:         $(value, count) \leftarrow scan\ q\ in\ region\ and\ count\ values$
12:         $Map(value, count) \leftarrow insert\ (value, count)$
13:     end for
14:     $Stage1\_Table \leftarrow insert\ Map(value, count)$
15: end procedure
16: function COMPUTE STATISTICS STAGE 1($q$)
17:     $Stage\_1\_Table \leftarrow$ CREATE_TABLE($Rs\_id, q$)
18:     STAGE_1($q$)
19: end function
```

The RegionServers then execute a two stage procedure, creating for each one an intermediate system table. As depicted in Algorithm 6, the first stage triggers an action in each RegionServer that locally scans the requested qualifier, and analyzes the cardinality of the distinct elements. This is done as each value may afterwards become a logical partition if a WF query requests an execution that partitions data according to the requested qualifier.

Moreover, it also registers the unique RegionServer identifier that produced each result, creating a map between the qualifier cardinalities and their location. The second stage is depicted in Algorithm 7. It triggers one of the available Region Servers to process the results from the first stage. Merging the results will identify which RegionServer should process each logical partition by comparing the registered cardinality of each distinct value read. For each value read, the RegionServer holding the largest cardinality will be elected as the optimum destination to process that value and ultimately, the logical partitions created by it. The results from computing the second stage are also kept in a system table in HBase.

During execution of a WF query, the shuffler mechanism will consider the pro-

Algorithm 7 CoProcessor update statistics stage 2

```
1: Initially:
2: $Stage\_1\_Table \leftarrow RegionServer$
3: $RegionServer\_List \leftarrow Region\ ServerRegions$
4: procedure CREATE_TABLE($RegionServer\_List$)
5:     $RegionServer \leftarrow add\ table\ with\ columns\ from$
6:                         $RegionServer\_List$
7: end procedure
8: procedure STAGE_2($Stage\_1\_Table$)
9:     for each $row\ \in\ Stage\_1\_Table$ do
10:            $(q, Rs\_1\ ...\ Rs\_n) \leftarrow row$
11:            $Rs\_max\_id \leftarrow largest\ (q, Rs\_1, Rs\_2, Rs\_n)$
12:     end for
13:     $Stage2\_Table \leftarrow insert\ (q, Rs\_max\_id)$
14: end procedure
15: function COMPUTE STATISTICS STAGE 2($q$)
16:     STAGE_2($q$)
17: end function
```

duced statistics and partition qualifier to obtain the assigned remote destination for the logical partition being computed. In order not to observe performance penalties associated with acquiring the destination for each row from HBase, the shuffling mechanism uses a Bloom Filter (BF) structure [Clearspring, 2017] configured to produce less than 5% of false positives to represent each available RegionServer. This structure is loaded into memory during the prepare statement stage of a query and it is kept in the HDQE memory for a configurable period of time. Testing each BF structure will find the row's destination by obtaining a positive answer from one of the BFs. If a BF produces a false positive reading and incorrectly forwards one row, the receiving remote destination will simply relay the row to where it thinks is the correct destination. Therefore, the probability of not being able to place a given row in the correct execution location, is bound to the probability of acquiring as many false positive readings as RegionServers, which becomes very small as the system size grows.

5.3.3 Shuffling Middleware

Shuffle operators are implemented over a communication middleware, enabling efficient intra-query synchronization and data exchange [Gonçalves et al., 2016],

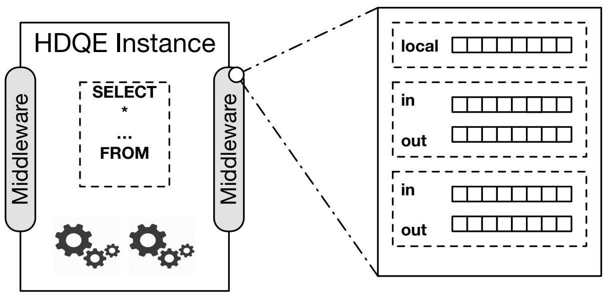


Figure 5.6: Structure of the shuffling queues.

and enabling intra-query parallelism for stateful operators. The middleware allows the distribution of data to multiple nodes through the establishment of a non-blocking socket-based overlay that interconnects available instances. The shuffle operator follows a push-based approach. When forwarding a row to a remote destination, the shuffler immediately attempts to deliver it. Each middleware instance holds a set of shuffling queues that enables it to asynchronously receive messages as depicted in Figure 5.6.

When a parallel connection is requested, the non-blocking communication sockets are established between HDQE instances along with the input and output buffers. As rows are pushed to the shuffler, the middleware verifies if the shuffling queues received rows from other remote instances. When a row is received its ideal destination is collected based on the partitioning clause requested by the query. If this row is meant to be locally computed, the middleware delivers it to the local incoming buffer to reach the local HDQE instance. If the row is meant to be sent to a remote location, the shuffler consults the BF structure to obtain the ideal remote location. The row is then serialized to the outgoing buffer of the remote location and written to the socket channel. In case the shuffler operator has no rows to forward to remote locations, it reads from its child operator (in this case the local storage) as dictated by the query plan, delivering them to the local HDQE instance. The shuffler remains active and does not block provided that further local rows are available to be read.

5.3.4 Implementation

The mechanisms presented were implemented in the HDQE. The engine was extended with the ability to interpret WF, particularly targeting the execution with ranking analytical operators. The engine is based on the Apache Derby project, which along all extra components described in section 5.1 build the Java based Hybrid Query Engine considered, a complex system assembled from custom and *off-the-shelf* components. The parallelism provided by the HDQE follows the `VOLCANO` [Graefe, 1990, 1993] plan-based approach, where the query operators themselves are oblivious to parallelism. Data is exchanged between instances and the operators in each instance are orchestrated to process disjoint sections of the workload. The required changes involved introducing new parsing options in the SQL parser, but also to customize the optimizer paths of such analytical queries.

The statistical collection and processing mechanisms were directly introduced atop the data substrate servers as HBase CoProcessors. This framework does not require any changes to HBase, as the CoProcessor itself is made available to all RegionServers as an external library that is loaded during RegionServer boot time.

To take advantage of the distributed nature of the HDQE's architecture, the query execution plans were modified to introduce the shuffling mechanism, allowing to distribute data rows across active HDQE instances. The communication middleware is based on a naive approach regarding data locality, simply considering the hash of the partitioning key as way to choose the remote destination of data. The modified communication middleware considers the statistical mechanism to actively forward data rows during query execution time by taking into account the statistical mechanism.

5.4 Evaluation

The HDQE alongside with the contributions introduced is evaluated in this section. We first start with an evaluation of the scalability of our proposal, by studying the impact of the statistical mechanism introduced. This was achieved by providing a fine grained evaluation that shows reduced variability across repetitions of the total execution time of a test query. From the same experiment we evaluated the total amount of data transfered during the scale out of the number of HDQE instances. We show the impact of handling different sized partitions on total query execution time and compare our proposal with Cloudera Impala,

```
select rank() OVER( PARTITION BY ss_customer_sk ORDER BY
        ss_store_sk) from store_sales
```

Figure 5.7: WF query for evaluation.

a leading cloud-based system for analytics over Hadoop.

5.4.1 Experimental Setup

The experiments considered a variable set of nodes with of either one of two configurations, namely *(i)* an Intel i3-2100-3.1GHz 64 bit processor with 2 physical cores (4 virtual), 8GB of RAM memory and 1 SATA II (3.0Gbit/s) 7200 RPM hard drive or *(ii)* an Intel i3-4170 3.7 GHz 64 bit processor with 2 physical cores (4 virtual), 8 GB of RAM and 1 SSD hard drive. All machines ran Ubuntu 14.04 LTS as the operating system and are interconnected by a switched Gigabit Ethernet network, deployed to provide a one hop connection between all hosts. In each configuration, one node provides the coordination service (Zookeeper), while another node holds the remaining coordination services, particularly: the HDFS Namenode, HBase Master and the HDQE's Transaction Manager and system loggers for fault tolerance. Each of the remaining nodes co-locate one instance of the HDFS Datanode, HBase RegionServer and HDQE. For every experiment we considered HBase 0.98. For every node, each component (HDFS Datanode, HBase RegionServer, HDQE instance and the operating system) was allocated with 2GB of main memory.

5.4.2 Performance and Scalability

The benchmarking suites from TPC are amongst the ones with the largest user base. From TPC's standards we chose TPC-DS, as it is the only one that considers WF in its queries. We considered the basic structure of one of TPC-DS's queries and extracted one sub-query holding a WF. From query `Q51` we extracted the sub-query structure holding a WF with a partitioning and ordering criterion, such as the ones considered in the rest of this chapter. For all experiments, we considered the query defined in Figure 5.7.

In all experiences, number of HDQE instances (each one with a single worker), does not take into account the two nodes reserved for coordination services. For

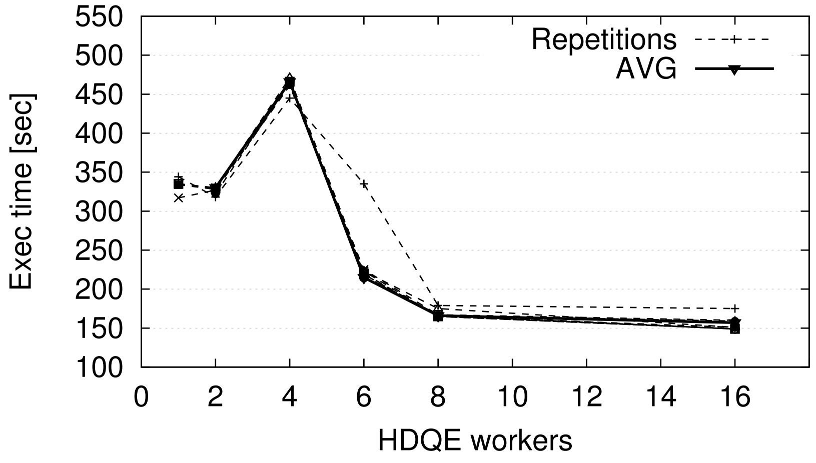


Figure 5.8: Comparison of execution time for the naive technique across runs.

the following experiments the database was populated with the `STORE_SALES` relation from TPC-DS, configured for a scale factor of 50GB. For each test, 10 independent runs (repetitions) were considered. Prior to query execution, the HDQE was instructed to activate the statistical mechanism by analyzing the qualifier set as the partitioning clause in the considered demo query, the `ss_-customer_sk` qualifier.

The first experiment explores the execution time for the WF query along with its variability. This was done for the naive and holistic techniques. On one hand, in the Naive mode, the query plan does not consider the statistical mechanism. Instead, the shuffler component produces the destination for a given row by producing the hash result of the value read from the qualifier controlling the partitioning. On the other hand, in the Holistic mode, the query plan considers the shuffling mechanism that is fed with data from the processed statistics, thus shifting data rows among nodes according to the partitioning criterion considered.

Figure 5.8 depicts the results for the Naive technique. It depicts all 10 repetitions (the dashed lines) and the computed average (solid line). We started by assessing the query execution time by resorting to a single HDQE instance and therefore considering no parallelism. We afterwards considered an increasing number of workers, namely: 2, 4, 6, 8 and 16, and conducted the following analysis based on the average. With no parallelism, one single HDQE instance was able to execute the example query in an average of 344 seconds. Overall, as we scale the number of HDQE workers we start to observe a reduction of execu-

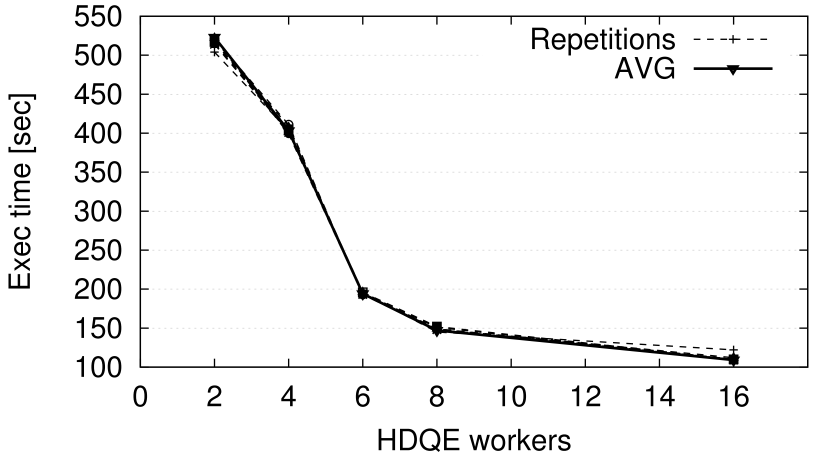


Figure 5.9: Comparison of execution time for the holistic technique across runs.

tion time. With 2 HDQE workers, the execution time slightly decreases to 327 seconds. However, with 4 HDQE workers the execution spikes and takes more than 450 seconds to execute the query. This is explained by simply hashing the values from the partitioning qualifier promoted a large shift of data to a subgroup of the available HDQE workers, imposing an imbalance in data distribution. As some HDQE workers have to deal with such an imbalance, the uneven number of rows to be processed causes all CPU bound stages of the query plan (sorting and computing the aggregation) to contribute to a higher execution time. As more HDQE workers are added, the data imbalance decreases. From 6 workers onwards, the registered execution times kept dropping with the increase in the amount of workers, namely 231 seconds with 6 workers, 168 seconds with 8 workers and 158 seconds with 16 workers. Overall, scaling the naive technique promoted a performance gain of 52%. Moreover, it is also possible to observe a standard deviation of 11.57% across all performed repetitions.

Figure 5.9 depicts the results for the Holistic technique we propose. This experiment considered all 10 repetitions and the computed average. We do not provide a data point for the absence of parallelism as the previous reading stands for the current experiment. The remaining experiments followed the same configuration as in the previous setup. Overall, we were able to sustain a significant decrease in query execution time by scaling the number of workers involved. With 2 workers, the parallel execution took 517 seconds to complete, registering the

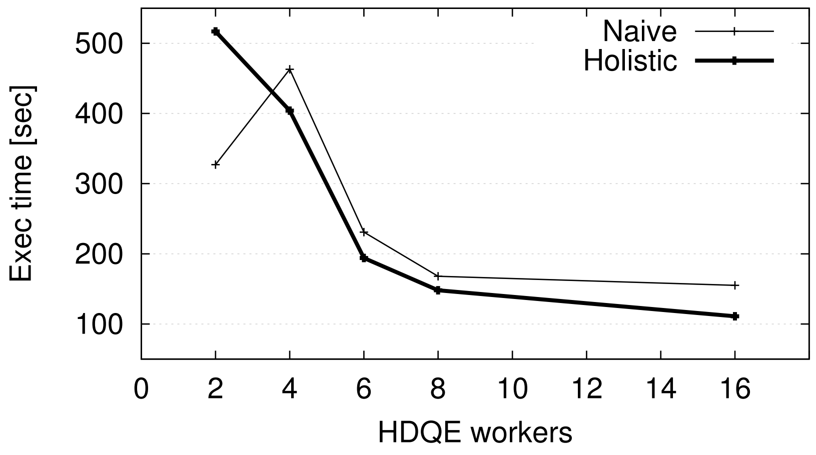


Figure 5.10: Comparison of execution time for the naive and holistic techniques.

highest value from our result set. The following configurations observed a steep decrease in execution time, decreasing to 404 seconds. Scaling to 6 workers, configured the largest decrease to 194 seconds, which represents a performance gain of 2 times when compared with the previous configuration. With 8 and 16 workers we were able to lower even more the execution time respectively to 148 and 111 seconds. Overall, scaling the holistic technique rendered a performance gain of 78% and a standard deviation of 3.16% across runs.

Moreover, we observed a lower variability between all repetitions. This is directly connected to the determinism provided by the statistical mechanism considered. The holistic technique is able to chose the ideal location for each logical partition, selecting the location that *a priori* holds more data of the logical partition being handled. This is not taken into account in the naive mechanism, which justifies the results.

Figure 5.10 presents the comparison between both techniques as we scale up the number of HDQE workers. Figure 5.11 renders the achieved performance gain between techniques across configurations. Overall, with the exception for the configuration with 2 workers, the Holistic mechanism was able to reduce execution time in all remaining configurations, being 17% faster on average. Considering the configuration with 2 workers, the Naive technique achieved a result 57% faster. That is, with only 2 workers, the Holistic mechanism introduces a large penalty by having to consult the BF structure to determine the ideal destination.

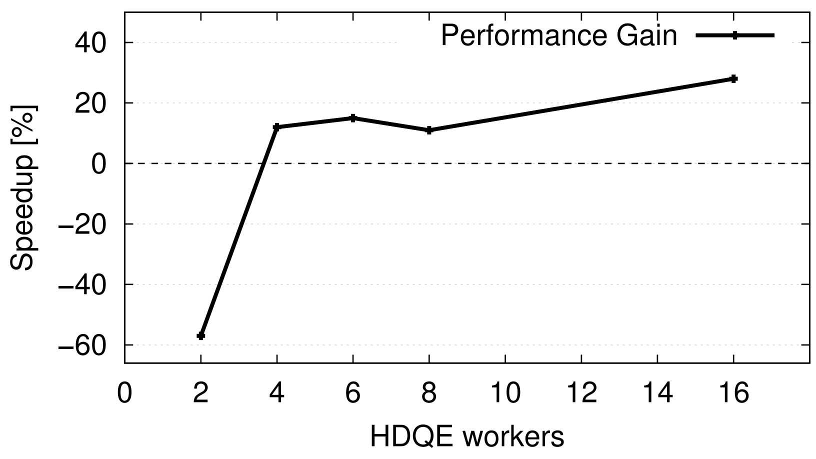


Figure 5.11: Performance gain for the holistic technique.

In comparison, the Naive technique only produces the hash for the value in the partitioning qualifier and pushes the data to be forwarded. From 2 workers onwards, the Holistic technique was always able to turn out a lower execution time when compared with the Naive technique. The performance gain analysis also shows that when we scale the number of workers beyond 2, we are always able to be at least 10% more efficient. Moreover, we were able to reach 27% performance gain when scaling up to 16 workers, reducing query execution time to 111 seconds. We did not scale our system above 16 workers, as the analysis of Figure 5.10 renders a marginal performance gain beyond that point.

5.4.3 Partition Sizes

To assess the impact that modifying the number of partitions and the size of each partition had on query execution time, we changed the distribution of logical partitions within the considered column qualifier. For this particular qualifier, TCP-DS's distribution creates logical partitions composed of 221 data rows. To perform this experiment, we considered the previous workload and modified the distribution considered for the `ss_customer_sk` qualifier. We traded the distribution considered by TPC-DS for a custom one. That is, we changed the `ss_customer_sk` qualifier and assigned it with a constant partition size throughout the entire qualifier, namely with 1 thousand and 1 million items. Thus, the number of logical partitions in a given configuration will change accordingly.

Partition size	**221**	1000	1000000
# Partitions	**651.583**	144.000	144

Table 5.1: Number of generated partitions per configuration. The items in bold depict the original distribution.

Table 5.1 details the number of generated logical partitions according to the selected partition size. For this experiment, we used the 16 worker configuration.

Regarding `STORE_SALES`'s `ss_customer_sk` qualifier, TPC-DS establishes a fixed size number of elements per partition. Increasing the benchmark's scale factor does not result in an increase in the partition size, but rather an increase in the total amount of partitions observed.

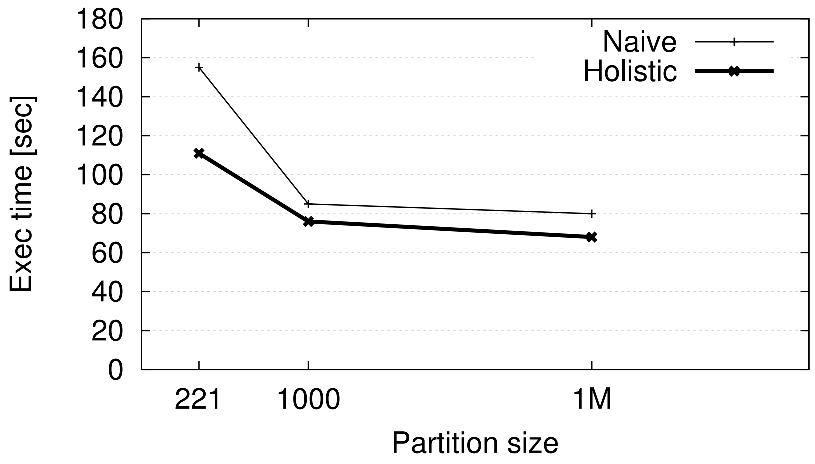


Figure 5.12: Partition size impact in query execution time. The horizontal axis is presented in logarithmic scale.

The results depicted in Figure 5.12 show that by increasing the partition size, the query total execution time decreased in both considered techniques. The Holistic technique proved to be on average 18% better across all tested configurations. The most favorable result was achieved for partitions with 1 million elements, which generated 144 partitions and the lowest registered execution time of 68 seconds.

Overall, this experiment shows how partition sizing is particularly relevant for the execution of this class of analytical functions. The considerations regarding the ideal distribution are obviously directly related to the number of parallel

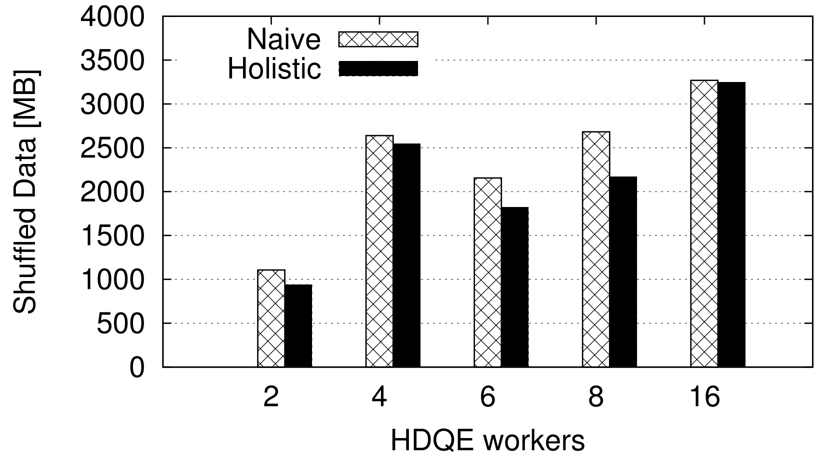


Figure 5.13: Shuffled data between HDQE workers.

workers considered. The results allow us to conclude that for achieving the best results, the data distribution should favor having a higher partition size.

5.4.4 Shuffled Data

From the previous experiments, we measured the data that was transfered through the shuffling mechanism. We considered the same hosts and workload as in the previous experiments and varied the number of workers. The results are depicted in Figure 5.13.

The results show that across all evaluated configurations, the Holistic mechanism was able to reduce the amount of data that had to be forwarded as opposed to the Naive technique. Overall, this result came up to 8%, on average, being more pronounced for the 8 worker configuration, with a decrease of 19%. For the configuration with 16 workers, the savings introduced by the Holistic technique were marginal, reaching 1%.

These results are a direct outcome of both the data workload, whose primary key influences the data placement on each HBase RegionServer, but also, of the distribution in each qualifier, particularly the one selected in the *Partition by* clause. As the number of workers grows, the number of row redirects (when a row is forwarded to one worker and needs to forwarded again) can also grow. This is a consequence from the use of Bloom Filters. Nevertheless, this can be adjusted by tuning the false positive configuration of the Bloom Filters, trading precision for memory in the HDQE.

5.4.5 Comparison with Cloudera Impala

Next, we provide a comparison between the HDQE and Cloudera Impala [Kornacker et al., 2015], an open-source SQL query engine architected for the Hadoop ecosystem. We chose Impala since it is widely used and shares a similar architecture with the HDQE. That is, Impala is built from several daemons that act as query engines, and a group of coordination daemons.

The architecture of the HDQE uses HBase as its data store, proving OLTP and OLAP capabilities. In both cases, data is stored in HBase.

In this experiment we intended to establish a comparison between both systems, considering only the OLAP performance. We considered a scenario with a 16 worker configuration. The HDQE was deployed as in Section 5.4.1. We considered the query depicted in Figure 5.7 and the same dataset built TPC-DS's `STORE_SALES` table.

Impala requires two coordination daemons, namely: Impala's StateStore, responsible to track the health of Impala's daemons and Impala's Catalog, responsible for relaying metadata changes across the cluster. Both were deployed in the same machines considered for the HDQE's coordination services. Moreover, metadata from SQL tables and statements are managed through Apache Hive [Thusoo et al., 2010], which is a mandatory dependency. Likewise, Hive was deployed a third coordination host.

The results depicted in Figure 5.14 show that the HDQE performance is superior in both configurations. The Naive HDQE configuration was able to be 28% better, while the Holistic HDQE configuration was able to reduce the query execution time by 48%. Impala achieves worse results as it was unable to evenly distribute data processing along its active daemons. Moreover, it relies in Hive to retrieve metadata regarding data location and data schema translation between SQL and the data store.

These results corroborate the effectiveness from the proposed technique. These techniques are particularly relevant when is not possible to change the database's schema in order to adopt a data distribution scheme that would benefit a particular workload, as the dynamism of the workload renders no ideal data partitioning scheme (*i.e.,* queries consider a diversity of different partitioning attributes and

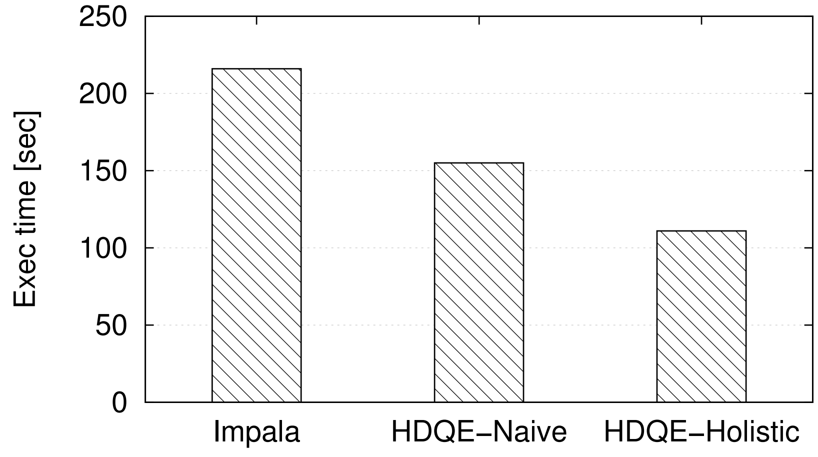


Figure 5.14: Comparison of execution time for the HDQE and Cloudera Impala.

change constantly).

5.4.6 Hybrid Performance

The previous experiments focused on the evaluation of the HDQE's performance regarding the OLAP only operation. To assess the capability of the HDQE to execute hybrid workloads, this subsection presents an experiment where we considered the hybrid workload induced by HTAPBench. First, we revisit subsection 3.4.3 where HTAPBench demonstrated its effectiveness to assess a hybrid database, where in fact the hybrid database considered was the HDQE described in this chapter. Nevertheless it was configured without any of the improvements introduced that took shape as the statistical mechanisms covered. Figure 3.7 is afterwards revisited as Figure 5.15.

This experiment was configured in a distributed environment comprised of 10 commodity hosts, 9 of which are responsible for handling and storing data, and the remaining node provides coordination and other global services, such as the Zookeeper ensemble, transactional manager and loggers. Each node is equipped with an Intel i3-2100-3.1GHz 64 bit processor with 2 physical cores (4 virtual), 8GB of RAM memory and 1 SATA II (3.0Gbit/s) hard drive, running Ubuntu 12.04 LTS as the operating system and interconnected by a switched Gigabit Ethernet network, with a 1 hop max distance in-between hosts. The following

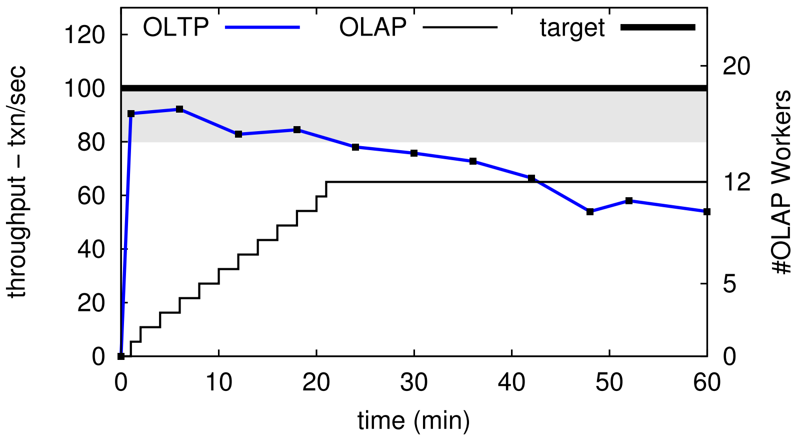


Figure 5.15: HTAPBench's evaluation of the HDQE.

```
SELECT sum(ol_amount) AS revenue FROM app.order_line WHERE
ol_delivery_d >= X AND ol_delivery_d < Y AND ol_quantity BETWEEN W
AND Z;
```

Figure 5.16: HTAPBench query `Q6`.

experiments consider the exact same configuration.

The HDQE was able to sustain the most balanced result in terms of the hybrid workload imposed by HTAPBench, in comparison with the OLTP and OLAP systems considered in subsection 3.4.3. However, as described in subsection 3.1, HTAPBench's OLAP portion of the workload is based on TPC-H. Thus, no WF analytical queries are provided. Therefore, HTAPBench's analytical workload was modified to introduce a set of WF as part of its analytical workload.

A group of analytical queries were selected to be changed and afterwards power the assessment of the effectiveness of the mechanisms introduced in a hybrid workload. The selected queries followed a criterion where the original TPC-H queries comprehended cumulative or ranking aggregations (*e.g.,* `Q1`, `Q3`, `Q5`, `Q6`, `Q9`, `Q11`, `Q18` or `Q19`). The full list of modifications is provided in the appendixes. As an example, we take into consideration query `Q6`, which is depicted in its original shape in Figure 5.16, as in HTAPBench, and the modified version holding WF in Figure 5.17.

Considering the previously introduced configuration, we verified the impact that the statistical mechanism had on the analytical portion of the hybrid work-

```
SELECT rank() OVER (PARTITION BY ol_d_id ORDER BY ol_amount) AS
    revenue FROM app.order_line WHERE ol_delivery_d >= X AND
        ol_delivery_d < Y AND ol_quantity BETWEEN W AND Z;
```

Figure 5.17: HTAPBench query `Q6` with WF.

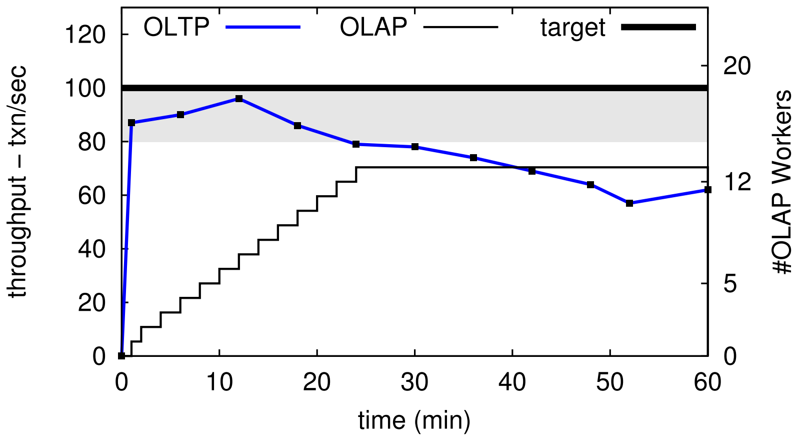


Figure 5.18: HDQE throughput evaluation.

load. HTAPBench was configured with an throughput threshold of 20% and a target throughput of 100 *tps*.

Figure 5.18 depicts results without considering the statistical mechanisms. This SUT was able to sustain 570 *tpmC* and 74 Qph, 21 of which held WF in their construction. These results are supported by 13 analytical streams, independently launched by HTAPBench's client balancer up to the 23*rd* minute, producing a *QpHpW* of 5.69 @ 570 *tpmC*.

Figure 5.19 depicts the results from executing the same workload but considering the mechanisms introduced to expedite the distributed execution of WF. This SUT was able to sustain 575 *tmpC* and 85 *Qph*, 30 of which held WF. The increase in the number of analytical queries executed was observed even though, the SUT was not able to deploy as many analytical workers as in the previous experiment.

The results from the previous experiments are compared in table 5.2. The results achieved for these experiments are in-line with the results in subsection 5.4.5, showing that when we the HDQE was equipped with the proposed statistical mechanism, a direct increase in the number of analytical queries was observed,

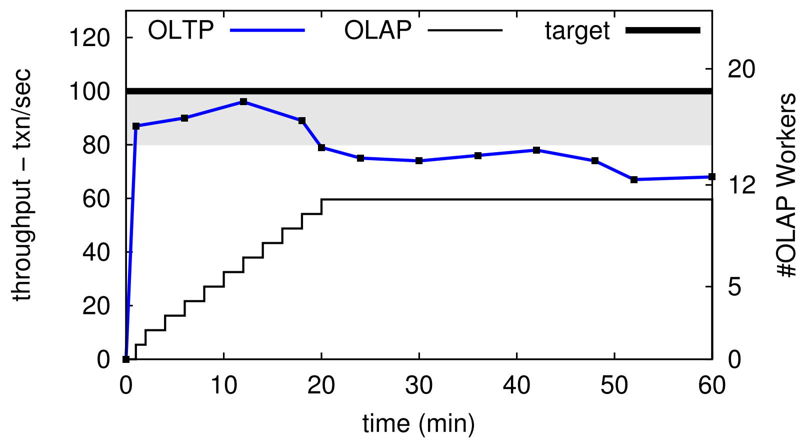


Figure 5.19: HDQE throughput evaluation with statistics.

	# OLAP workers	QpH	QpHpW
HDQE without statistics	13	74	5.69 @ 570
HDQE with Statistics	11	85	7.72 @ 575

Table 5.2: Analytical performance with and without statistics in a hybrid workload.

even in the presence of a hybrid workload.

5.5 Related Work

The Apache processing environment provides a set of tools that expedite the data manipulation process, offering solutions targeted to analyze data stored in distributed layouts. For instance, Hive [Thusoo et al., 2009, Graefe, 1993] provides OLAP data warehousing capabilities, offering the ability to execute WF but relying in ETL, bypassing real-time processing. Impala [Kornacker et al., 2015] offers a OLAP oriented SQL query engine that leverages the HDFS or HBase to deploy a set of distributed query engine daemons. Impala, is not engineered to account for Hybrid workloads, focusing only on read-mostly queries and bypassing transactional semantics. Even though these systems can be clustered together and theoretically use the same data, it is not clear what is the overall cost penalties or the issues that may derive from the concurrent execution of all components.

Nevertheless, Impala, due to its distributed architecture and ability to process WF is actually the best candidate for a direct comparison with our proposal, in a OLAP only configuration.

The fast-paced field of databases saw in recent years the inception of several new projects specifically engineered to take advantage of cloud-based infrastructures in order to scale. Particularly, the need for real-time analytics and the associated requirement for data freshness or real-time operational analytics, largely contributed to the development of new Hybrid database solutions [Larson et al., 2015, Plattner, 2009]. They merge the consistency requirements for transactional systems with the need to perform fast analytical operations on top of production data. For example, SAP HANA [SAP, 2014] used multi-version concurrency control (MVCC) mechanisms alongside with in-memory data structures which are periodically fed with deltas of the operational data. Microsoft SQL Server offers two specialized engines for OLTP and OLAP [Larson et al., 2015]. Oracle [Lahiri et al., 2015, Corporation, 2015] also keeps separate engines but considers in-memory versions, allowing for fast execution while relying in the same durable data, employing a publish-subscribe methods to propagate changes between them. Hyper [Kemper and Neumann, 2011] provides hybrid workload capabilities by using the fork primitive of the operating system to provide MVCC, enabling serializability for transaction processing along with snapshot isolation for analytical workloads. MemSQL [Shamgunov, 2014] provides in-memory hybrid workloads with distributed query processing capabilities. BatchDB [Makreshanski et al., 2017] provides a new system for hybrid workload that considered primary-secondary replication mechanism to feed replicas that specialize in either OLTP or OLAP operation, but does not offer full SQL compliance and therefore no WF capability.

5.6 Remarks

In this chapter, we addressed the need for parallel implementations of Window Functions in hybrid distributed databases. The architecture presented shares some of the principles considered in current state-of-the-art. It is built from modular and off-the-shelf components, pertaining a more flexible environment and taking advantage of the inherent optimizations offered by each single component. Our approach leverages statistics collected from the underlying data to guide the shuffling of rows among distributed workers in order to reduce data movement. This results, not only in reduced resource usage, but also in improved execution time of analytical workloads built from queries with WF.

Our extensive evaluation showed that the considered architecture, together with the proposed mechanisms achieved an improvement of over 17% when compared to a naive approach, oblivious to data distribution and an improvement of over 38% when compared to a state of the art approach such as Cloudera's Impala. Moreover, when considering the benchamarking suite proposed in chapter 3, HTAPBench, the proposed mechanisms introduced analytical throughput gains in the order of 14% during a hybrid workload.

Chapter 6

Conclusion

In this dissertation, we focused on three main challenges associated with the provision of data analytics over a HTAP database system, namely: how to assess a HTAP system, how to design analytical functions in a hybrid context and how to take advantage of current distributed database architectures to expedite such processing.

Hybrid database systems are intrinsically related to *real-time* data analytics, that is, systems that are capable of producing aggregations that merge historical data and the result of real-time data streams holding transactional data. This has been the key design choice toward eschewing the ETL process. A process that introduces considerable delays in the data analysis cycle, being responsible for acquiring, transforming and loading data from a OLTP to a OLAP database. The demand for such process arises from two distinct data schemes with incompatible goals. To shorten this gap, HTAP systems often consider distinct engines and their underlying data layouts, loading the database application with the responsibility to transparently provide the client with a hybrid view of the database [Özcan et al., 2017].

To gain insight about which approaches might or not render the best results in a HTAP system, we introduced a new benchmarking system, HTAPBench, that addresses Gartner's recommendations [Pezzini et al., 2014]. This need emerged from the misfit of current benchmarking approaches, but also, from the need to comprehend both the transactional and analytical stages of a hybrid workload in a unified metric, capable of enabling the quantitative comparison between systems. The action in HTAPBench is centered around a client balancer that controls the transactional activity and manages the execution of an increasing amount of analytical workers, while providing homogeneous and comparable results across executions. HTAPBench was validated on top of OLTP, OLAP and HTAP systems, demonstrating its expressiveness to characterize such systems. Moreover, HTAPBench is able to exercise the underlying storage layout as expected for each workload type. The results allow to conclude that by using the proposed approach, we were able to introduce the required workload randomness while keeping the results comparable, ensuring equal query execution costs across the whole dataset.

To account for the singularities of analytical functions with a HTAP system, we focused on a specific class of analytical functions, designated as *Window Functions*. Window Functions are particularly appealing for scenarios with OLAP or HTAP systems, allowing to conduct time series analysis or computing aggregations over configurable data frames, customizing them to reflect newly ingested items from ETL or real-time production data. Leveraging distributed scenarios, often considered in both these systems, we introduced two novel strategies for the parallel execution of window functions, enabling several workers to split and share query execution, while choosing the optimum execution location that minimizes data forwarding among workers. This took shape in a mechanism that introduces a holistic awareness regarding data execution of this class of analytical functions. Moreover, we built on this mechanism to optimize the way parallel workers exchange data among peers, exploring similarity between data partitions in the WF construction.

The presented techniques leverage the distributed and highly scalable PaaS considered. We introduced the ability to interpret WF and equipped the underlying data store and QE with the proposed statistical mechanisms. All parallel opportunities were explored and together with the considered strategies, we were able to show through our extensive evaluation presented an improvement of 17% over the baseline system and, over 38% when compared with a state of the art approach. Moreover, we were able to demonstrate the expressiveness of the WF environment in the context of a hybrid database system, capable of powering *real-time* data analytics.

6.1 Future Work

As we build on the main problems discussed in this dissertation, and on the solutions proposed, it is possible to derive a new set of challenges. In the following we briefly discuss such possibilities toward future research directions.

First, the mechanisms considered for the acquisition of statistical markers may be exported to other aspects within the distributed execution of analytical operators. In fact, the techniques comprehended in this dissertation are much broader than their application in the scope of *Window Functions*, being also applicable to other operators that are built around grouping or other types of logical partitioning. Moreover, it is also possible to employ this statistical characterization to guide the load balancing of data, adjusting the physical location of data partitions according to the workload.

Second, as the underlying architectures shift to offer a solo in-memory operation, it is still possible to consider the holistic awareness to guide and improve the parallel processing capabilities. To take this concept to fruition, we envision to use low latency network interconnects as part of the system architecture, enabling to exchange data directly between the main memory registries of distinct remote nodes of the database.

Bibliography

Reactive programming. http://reactivex.io, 2015. - **Cited** on page 72.

Reactive programming for java. https://github.com/ReactiveX/RxJava, 2015. - **Cited** on page 72.

Daniel Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreos, Samuel Madden, et al. The design and implementation of modern column-oriented database systems. *Foundations and Trends® in Databases*, 5(3):197–280, 2013. - **Cited** on page 16.

Apache. The apache derby project. Technical report, Apache Foundation, 2016. URL `https://db.apache.org/derby/derby_charter.html`. - **Cited** on page 82.

Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394. ACM, 2015. - **Cited** on page 21.

C Bange, T Grosser, and N Janoschek. Big data survey europe: Usage, technology and budgets in european best-practice companies. *White Paper, BARC*, 2013. - **Cited** on page 3.

Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. *SIGMOD Rec.*, 24(2), May 1995. ISSN 0163-5808. - **Cited** on pages 12, 21 and 83.

Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency* *control and recovery in database systems.* 1986. ISBN 0-201-10715-5. - **Cited** on page 12.

Anja Bog, Jens Krüger, and Jan Schaffner. A composite benchmark for online transaction processing and operational reporting. In *Advanced Management of Information for Globalized Enterprises, 2008. AMIGE 2008. IEEE Symposium on*, pages 1–5. IEEE, 2008. - **Cited** on page 50.

Anja Bog, Hasso Plattner, and Alexander Zeier. A mixed transaction processing and operational reporting benchmark. *Information Systems Frontiers*, 13(3): 321–335, July 2011a. ISSN 1387-3326. doi: 10.1007/s10796-010-9283-8. URL `http://dx.doi.org/10.1007/s10796-010-9283-8`. - **Cited** on page 50.

Anja Bog, Kai Sachs, and Alexander Zeier. Benchmarking database design for mixed oltp and olap workloads. In *Proceedings of the 2Nd ACM/SPEC International Conference on Performance Engineering*, ICPE '11, pages 417–418, New York, NY, USA, 2011b. ACM. ISBN 978-1-4503-0519-8. doi: 10.1145/1958746.1958806. URL `http://doi.acm.org/10.1145/1958746.1958806`. - **Cited** on page 50.

Anja Bog, Kai Sachs, and Hasso Plattner. Interactive performance monitoring of a composite oltp and olap workload. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 645–648, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1247-9. doi: 10.1145/2213836.2213921. URL `http://doi.acm.org/10.1145/2213836.2213921`. - **Cited** on page 50.

Dhruba Borthakur et al. Hdfs architecture guide. *Hadoop Apache Project*, 53, 2008. - **Cited** on page 21.

Paul G Brown and Peter J Hass. BHUNT: Automatic discovery of fuzzy algebraic constraints in relational data. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 668–679. VLDB Endowment, 2003. - **Cited** on page 70.

Michael J Cahill, Uwe Röhm, and Alan D Fekete. Serializable isolation for snapshot databases. *ACM Transactions on Database Systems (TODS)*, 34(4):20, 2009. - **Cited** on page 13.

Yu Cao, Chee-Yong Chan, Jie Li, and Kian-Lee Tan. Optimization of analytic window functions. *Proceedings of the VLDB Endowment*, 5(11):1244–1255, 2012. - **Cited** on pages 54, 55 and 79.

Biswapesh Chattopadhyay, Liang Lin, Weiran Liu, Sagar Mittal, Prathyusha Aragonda, Vera Lychagina, Younghee Kwon, and Michael Wong. Tenzing a sql implementation on the mapreduce framework. In *Proceedings of VLDB*, pages 1318–1327, 2011. - **Cited** on page 15.

Clearspring. Clearspring analytics library. Technical report, Clearspring, 2017. URL `https://github.com/addthis/stream-lib`. - **Cited** on page 90.

Richard Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi Kuno, Raghunath Nambiar, Thomas Neumann, Meikel Poess, et al. The mixed workload ch-benchmark. In *Proceedings of the Fourth International Workshop on Testing Database Systems*, page 8. ACM, 2011. - **Cited** on pages 27 and 50.

Microsoft Coporation. Transact-SQL. Technical report, Microsoft Corporation, 2013. URL `https://msdn.microsoft.com/library/ms189461(SQL.130).aspx`. - **Cited** on pages 53 and 79.

Oracle Corporation. SQL analysis and reporting. Technical report, Oracle Corporation, 2015. URL `http://docs.oracle.com/database/121/DWHSG/analysis.htm#DWHSG8659`. - **Cited** on pages 53, 79, 81 and 105.

Standard Performance Evaluation Council. *Standard Performance Evaluation Council.* 2015a. URL `https://www.spec.org/`. - **Cited** on page 48.

Storage Performance Council. *Storage Performance Council.* 2015b. URL `http://www.storageperformance.org/home/`. - **Cited** on page 48.

Transaction Processing Performance Council. *TPC Benchmark C.* 2010a. URL `http://www.tpc.org/tpcc/spec/tpcc_current.pdf`. - **Cited** on pages 23, 28 and 48.

Transaction Processing Performance Council. *TPC Benchmark H.* 2010b. URL `http://www.tpc.org/tpc_documents_current_versions/pdf/tpch2.17.1.pdf`. - **Cited** on pages 23 and 49.

Transaction Processing Performance Council. *TPC Benchmark DS*. 2012. URL `ttp://www.tpc.org/tpcds/spec/tpcds_1.1.0.pdf`. - **Cited** on pages 27 and 49.

Transaction Processing Performance Council. *The Transaction Processing Performance Council*. 2015c. URL `http://www.tpc.org/`. - **Cited** on page 48.

Transaction Processing Performance Council. *TPC Benchmark E*. 2015d. URL `http://www.tpc.org/tpc_documents_current_versions/pdf/tpce-v1.14.0.pdf`. - **Cited** on pages 23 and 49.

Frederica Darema. The SPMD model: Past, present and future. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 2131. Springer Berlin Heidelberg, 2001. doi: 10.1007/3-540-45417-9_1. - **Cited** on page 85.

Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008. - **Cited** on pages 3 and 15.

Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *PVLDB*, 7(4):277–288, 2013. - **Cited** on page 36.

Byron Ellis. *Real-time analytics: Techniques to analyze and visualize streaming data*. John Wiley & Sons, 2014. - **Cited** on page 3.

Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. *ACM Transactions on Database Systems (TODS)*, 33(2):6, 2008. - **Cited** on page 70.

Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making snapshot isolation serializable. *ACM Transactions on Database Systems (TODS)*, 30(2):492–528, 2005. - **Cited** on page 13.

Clark D. French. "one size fits all" database architectures do not work for dss. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, pages 449–450, New York, NY, USA, 1995. ACM. ISBN 0-89791-731-6. doi: 10.1145/223784.223871. - **Cited** on page 24.

Florian Funke, Alfons Kemper, and Thomas Neumann. Benchmarking hybrid oltp&olap database systems. In *BTW*, pages 390–409, 2011. - **Cited** on page 50.

Hector Garcia-Molina. *Database systems: the complete book*. Pearson Education India, 2008. - **Cited** on pages 25 and 66.

Minos Garofalakis, Daniel Keren, and Vasilis Samoladas. Sketch-based geometric monitoring of distributed stream queries. *Proc. VLDB Endow.*, 6(10):937–948, August 2013. ISSN 2150-8097. doi: 10.14778/2536206.2536220. URL `http://dx.doi.org/10.14778/2536206.2536220`. - **Cited** on page 80.

Lars George. *HBase: The Definitive Guide: Random Access to Your Planet-Size Data*. O'Reilly Media, Inc., 2011. - **Cited** on pages 21, 63, 81, 82 and 88.

Rui C Gonçalves, José Pereira, and Ricardo Jiménez-Peris. An RDMA middleware for asynchronous multi-stage shuffling in analytical processing. In *Distributed Applications and Interoperable Systems*, pages 61–74. Springer, 2016. - **Cited** on pages 69 and 90.

Goetz Graefe. *Encapsulation of parallelism in the Volcano query processing system*, volume 19. ACM, 1990. - **Cited** on page 92.

Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys (CSUR)*, 25(2):73–169, 1993. - **Cited** on pages 92 and 105.

Jim Gray. *Benchmark handbook: for database and transaction processing systems*. Morgan Kaufmann Publishers Inc., 1992. - **Cited** on page 23.

Hadoop. ORC reference, 2017a. URL `https://orc.apache.org`. - **Cited** on page 21.

Hadoop. Parquet reference, 2017b. URL `https://parquet.apache.org`. - **Cited** on page 21.

Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, 1983. ISSN 0360-0300. doi: 10.1145/289.291. URL `http://doi.acm.org/10.1145/289.291`. - **Cited** on pages 11 and 48.

Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, volume 8, page 9. Boston, MA, USA, 2010. - **Cited** on page 82.

IBM. OLAP specification. Technical report, IBM, 2013. URL `http://www.ibm.com/support/knowledgecenter/SSEPGG_10.5.0/com.ibm.db2.luw.sql.ref.doc/doc/r0023461.html`. - **Cited** on pages 20, 53, 79 and 81.

IDC. Worldwide semiannual big data and analytics spending guide, 2015. URL `https://www.idc.com/getdoc.jsp?containerId=prUS41826116`. - **Cited** on page 1.

Ihab F Ilyas, Volker Markl, Peter Haas, Paul Brown, and Ashraf Aboulnaga. CORDS: automatic discovery of correlations and soft functional dependencies. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 647–658. ACM, 2004. - **Cited** on page 79.

Ricardo Jimenez-Peris, Marta Patiño-Martinez, Kostas Magoutis, Angelos Bilas, and Ivan Brondino. Cumulonimbo: A highly-scalable transaction processing platform as a service. *ERCIM News*, 89(null):34–35, 2012. - **Cited** on page 83.

Ricardo Jimenez-Peris, Marta Patino-Martinez, Bettina Kemme, Ivan Brondino, José Pereira, Ricardo Vilaça, Francisco Cruz, Rui Oliveira, and Yousuf Ahmad. Cumulonimbo: A cloud scalable multi-tier sql database. *Data Engineering*, page 73, 2015. - **Cited** on page 81.

David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663. ACM, 1997. - **Cited** on page 64.

Alfons Kemper and Thomas Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 195–206. IEEE, 2011. - **Cited** on pages 20 and 105.

Rusty Klophaus. Riak core: Building distributed applications without shared state. In *ACM SIGPLAN Commercial Users of Functional Programming*, page 14. ACM, 2010. - **Cited** on page 81.

Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, et al. Impala: A modern, open-source SQL engine for hadoop. In *CIDR*, volume 1, page 9, 2015. - **Cited** on pages 21, 53, 79, 100 and 105.

Tirthankar Lahiri, Shasank Chavan, Maria Colgan, Dinesh Das, Amit Ganesh, Mike Gleeson, Sanket Hase, Allison Holloway, Jesse Kamp, Teck-Hua Lee, et al. Oracle database in-memory: A dual format in-memory database. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pages 1253–1258. IEEE, 2015. - **Cited** on page 105.

Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010. - **Cited** on pages 21, 64 and 81.

Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. The vertica analytic database: C-store 7 years later. *Proceedings of the VLDB Endowment*, 5(12):1790–1801, 2012. - **Cited** on page 15.

Per-Åke Larson, Adrian Birka, Eric N Hanson, Weiyun Huang, Michal Nowakiewicz, and Vassilis Papadimos. Real-time analytical processing with sql server. *Proceedings of the VLDB Endowment*, 8(12):1740–1751, 2015. - **Cited** on page 105.

Kyong-Ha Lee, Yoon-Joon Lee, Hyunsik Choi, Yon Dohn Chung, and Bongki Moon. Parallel data processing with mapreduce: a survey. *AcM sIGMoD Record*, 40(4):11–20, 2012. - **Cited** on page 3.

Viktor Leis, Kan Kundhikanjana, Alfons Kemper, and Thomas Neumann. Efficient processing of window functions in analytical SQL queries. *Proc. VLDB Endow.*, 8(10):1058–1069, June 2015. ISSN 2150-8097. doi: 10.14778/2794367.2794375. URL `http://dx.doi.org/10.14778/2794367.2794375`. - **Cited** on page 79.

Hai Liu, Dongqing Xiao, Pankaj Didwania, and Mohamed Y. Eltabakh. Exploiting soft and hard correlations in big data query optimization. *Proc. VLDB Endow.*, 9(12):1005–1016, August 2016. ISSN 2150-8097. doi: 10.14778/2994509.2994519. URL `http://dx.doi.org/10.14778/2994509.2994519`. - **Cited** on page 70.

Xiufeng Liu, Nadeem Iftikhar, and Xike Xie. Survey of real-time processing systems for big data. In *Proceedings of the 18th International Database Engineering & Applications Symposium*, pages 356–361. ACM, 2014. - **Cited** on page 3.

Darko Makreshanski, Jana Giceva, Claude Barthels, and Gustavo Alonso. Batchdb: Efficient isolated execution of hybrid oltp+ olap workloads for interactive applications. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 37–50. ACM, 2017. - **Cited** on page 106.

Raghunath Othayoth Nambiar and Meikel Poess. The making of tpc-ds. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pages 1049–1058. VLDB Endowment, 2006. URL `http://dl.acm.org/citation.cfm?id=1182635.1164217`. - **Cited** on page 23.

Fatma Özcan, Yuanyuan Tian, and Pinar Tözün. Hybrid transactional/analytical processing: A survey. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1771–1775. ACM, 2017. - **Cited** on pages 20 and 107.

Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C Mowry, Matthew Perron, Ian Quah, et al. Self-driving database management systems. In *CIDR*, 2017. - **Cited** on page 20.

M Pezzini, D Feinberg, N Rayner, and R Edjlali. Hybrid transaction/analytical processing will foster opportunities for dramatic business innovation. *Gartner (2014, January 28) Available at https://www.gartner.com/doc/2657815/hybrid-transactionanalytical-processing-foster-opportunities*, 2014. - **Cited** on pages 4, 20, 24, 25, 27 and 107.

Hasso Plattner. A common database approach for oltp and olap using an in-memory column database. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 1–2. ACM, 2009. - **Cited** on page 105.

Meikel Poess and Chris Floyd. New tpc benchmarks for decision support and web commerce. *SIGMOD Rec.*, 29(4):64–71, December 2000. ISSN 0163-5808. doi: 10.1145/369275.369291. URL `http://doi.acm.org/10.1145/369275.369291`. - **Cited** on page 49.

Viswanath Poosala, Peter J Haas, Yannis E Ioannidis, and Eugene J Shekita. Improved histograms for selectivity estimation of range predicates. In *ACM SIGMOD Record*, volume 25, pages 294–305. ACM, 1996. - **Cited** on page 66.

Viswanath Poosala, Venkatesh Ganti, and Yannis E. Ioannidis. Approximate query answering using histograms. *IEEE Data Eng. Bull.*, 22(4):5–14, 1999. - **Cited** on page 66.

Dan RK Ports and Kevin Grittner. Serializable snapshot isolation in postgresql. *Proceedings of the VLDB Endowment*, 5(12):1850–1861, 2012. - **Cited** on page 12.

Postgresql. Advanced features - window functions. Technical report, Postgresql, 2015. URL `https://www.postgresql.org/docs/9.4/static/tutorial-window.html`. - **Cited** on pages 53 and 79.

Stephen Revilak, Patrick O'Neil, and Elizabeth O'Neil. Precisely serializable snapshot isolation (pssi). In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 482–493. IEEE, 2011. - **Cited** on page 12.

SAP. SAP HANA SQL reference, 2014. URL `https://help.sap.com/hana/SAP_HANA_SQL_and_System_Views_Reference_en.pdf?original_fqdn=help.sap.de`. - **Cited** on pages 20, 21, 53, 79, 81 and 105.

SAP. SAP Vora reference, 2017. URL `https://www.sap.com/products/hana-vora-hadoop.html`. - **Cited** on page 21.

Nikita Shamgunov. The memsql in-memory database system. In *IMDM@ VLDB*, 2014. - **Cited** on page 106.

Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010. - **Cited** on pages 2, 3, 15 and 83.

Michael Stonebraker. Technical perspective one size fits all: an idea whose time has come and gone. *Communications of the ACM*, 51(12):76–76, 2008. - **Cited** on page 2.

Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: A warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2 (2):1626–1629, August 2009. ISSN 2150-8097. doi: 10.14778/1687553.1687609. URL `https://doi.org/10.14778/1687553.1687609`. - **Cited** on page 105.

Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu, and Raghotham Murthy. Hive-a petabyte scale data warehouse using hadoop. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 996–1005. IEEE, 2010. - **Cited** on pages 15 and 100.

Jordan Tigani and Siddartha Naidu. *Google BigQuery Analytics*. John Wiley & Sons, 2014. - **Cited** on page 15.

Transaction Processing Performance Council. *TPC Benchmark DS*. 2012. URL `http://www.tpc.org/tpcds/spec/tpcds_1.1.0.pdf`. - **Cited** on pages 53 and 74.

Ricardo Vilaça, Francisco Cruz, José Pereira, and Rui Oliveira. An effective scalable sql engine for nosql databases. In *IFIP International Conference on Distributed Applications and Interoperable Systems*, pages 155–168. Springer, 2013. - **Cited** on page 83.

Florian M Waas. Beyond conventional data warehousing—massively parallel data processing with greenplum database. In *International Workshop on Business Intelligence for the Real-Time Enterprise*, pages 89–96. Springer, 2008. - **Cited** on page 15.

Calisto Zuzarte, Hamid Pirahesh, Wenbin Ma, Qi Cheng, Linqi Liu, and Kwai Wong. Winmagic: Subquery elimination using window aggregation. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 652–656. ACM, 2003. - **Cited** on pages 53, 55 and 79.