論文 / 著書情報
Article / Book Information

題目(和文)	高い鮮度で一貫性保証と更新集約適用を行う HTAPシステム高性能化に関する研究
Title(English)	A Study on Performance Enhancement of Highly Data-Fresh HTAP Systems Guaranteeing the Consistency and Aggregating Updates
著者(和文)	塩井隆円
Author(English)	Takamitsu Shioi
出典(和文)	学位:博士(工学), 学位授与機関:東京工業大学, 報告番号:甲第12816号, 授与年月日:2024年6月30日, 学位の種別:課程博士, 審査員:宮﨑 純,吉瀬 謙二,小林 隆志,渡部 卓雄,金子 晴彦
Citation(English)	Degree:Doctor (Engineering), Conferring organization: Tokyo Institute of Technology, Report number:甲第12816号, Conferred date:2024/6/30, Degree Type:Course doctor, Examiner:,,,,
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

A Study on Performance Enhancement of Highly Data-Fresh HTAP Systems Guaranteeing the Consistency and Aggregating Updates (高い鮮度で一貫性保証と更新集約適用を行うHTAPシステム高性能化に関する研究)

by

Takamitsu Shioi

Submitted to the Department of Computer Science, School of Computing in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Engineering at the TOKYO INSTITUTE OF TECHNOLOGY April 2024.

Thesis Committee:

Prof. Jun Miyazaki

Prof. Kenji Kise

Prof. Takashi Kobayashi

Assoc. Prof. Haruhiko Kaneko

Prof. Takuo Watanabe

Abstract

Hybrid transactional/analytical processing (HTAP) systems, developed to support online transactional processing (OLTP) and online analytical processing (OLAP), have attracted much attention. There are two types of processing in the database management system (DBMS), OLTP mainly for write processing and OLAP mainly for read processing of data. HTAP systems aim to execute OLAP without unduly affecting OLTP performance because the data updated in OLTP is required to quickly read out on the OLAP side and used for data analysis. Therefore, it is important for HTAP systems not to degrade conventional OLTP performance in the systems caused from participating OLAP workloads.

Concurrency control (CC) that ensures consistency of updated data is an essential element of OLTP systems. There are two types of HTAP system, called unified and decoupled storage architectures from the difference of physical isolated components of OLTP and OLAP. Unified storage HTAP systems can achieve serializability more easily than can decoupled storage architectures, whereas decoupled storage HTAP systems comprise separate OLTP-side and OLAP-side subsystems, aiming to achieve better individual OLTP and OLAP performances than those achieved by a unified storage system. Serializable isolation would be the ideal, but achieving serializability can cause aborts or waits in transaction processing and can introduce potential performance problems because the HTAP system's serializability requires additional OLAP read-only transactions to be included in its OLTP. Therefore, the decoupled storage HTAP system has become established, which uses snapshot isolation (SI) to address the issues around performance deterioration.

The initial aim of this study is to achieve serializability for decoupled storage HTAP systems such as replicated systems. Although aborts and waits of transactions are bound to occur if serializable isolation is to be guaranteed, a decoupled storage HTAP system will be more hampered by them than will a unified storage HTAP system. For HTAP workloads, the OLAP-side subsystem sends back read sets, which depend on the OLAP workload, to the OLTP side via bidirectional synchronous communication (such as two phase locking (2PL) for example), thereby increasing the communication costs between the OLTP and OLAP subsystems. Such OLTP-side subsystem plays a role of globally managing transactions in the regular way of distributed environment for applying HTAP systems. To avoid having to send a transaction's information back to the OLTP subsystem, this dissertation aims to achieve serializability by unidirectional communication from OLTP to OLAP. In the other words, my approach is to

achieve serializability without additional aborts and waits derived from read-only transaction participations. This study proposes using read-safe snapshots (RSS), developed using multiversion concurrency control (MVCC) theory, and introduce an RSS construction algorithm that utilizies serializable snapshot isolation (SSI). For the serializability of HTAP systems, the theoretical model makes use of multiversion and allows for more schedules with read operations whose corresponding write operations do not participate in dependency of transactions. The implementations of the algorithm uses PostgreSQL as an open-source database system that offers SSI. The algorithm was implemented and used to evaluate both unified and decoupled storage HTAP systems. The OLTP performance for a unified storage architecture was better by about 45%, without degrading OLAP performance, than SSI equipped with a read-only optimization method called Safe-Snapshots. For the decoupled storage architecture, the algorithm compared my system with a decoupled storage system that does not guarantee serializable isolation. The comparative system was configured that OLAP was executed under SI and OLTP was executed under SSI. My system limited performance deterioration to about 15%. In addition, the data-freshness performance, when new RSS became available, averaged 4.4 seconds, with the maximum delay time of 10.6 seconds, during a 30-minutes experiment.

Moreover, the second aim of this dissertation addresses implementation problems in HTAP. Due to the property of connecting OLTP and OLAP components, HTAP systems need to deal with the problem of different access methods for not deteriorating each OLTP and OLAP performances. OLAP accesses data represented in tabular form per column. To enhance the analytical performance, HTAP systems apply a column-oriented storage model to OLAP-side components for efficiently reading the columnar data to DRAM or CPU cache. The column-oriented storage model can compress and store the columnar data of same data type in non-volatile storage. The major issue in hybrid studies of OLTP and OLAP fields is the conversion from processed data in OLTP to columnar data for accelerating the whole processing. Since many methods exist in there, either commercial or academia DBMS, this study introduces a mechanism used by the tuple identifier (TiD) as a version identifier of data object based on MVCC such as RSS. In particular, an access unit of OLTP is based on MVCC and MVCC could have assumed the tuple-/row-oriented access based on the page model. The serializability of HTAP systems had not been particularly proposed but RSS in the first part of this dissertation can reasonably resolve it, which can become able to focus on the traditional problem of OLTP vs. OLAP access methods. This study approaches an architecture about the hybrid of OLTP and OLAP on the presupposition of using RSS.

This dissertation proposes a system that temporarily retains row-oriented data on DRAM

and the updated data are reflected in column-oriented data such as general column-oriented database systems. Because serializable isolation cannot be assured on the property of HTAP serializability, in the case that the versions recently updated by OLTP are reflected in OLAP results. The proposed system creates a TiD-based column update index to resolve the gap between row/column-access unit for real-time modification of columnar data on the DRAM and a storage device. The mechanism is able to efficiently hit the DRAM cache by using an algorithm computing the number of updated tuples in columnar data blocks such that the modifications can reduce disk I/O frequencies. These proposed methods were connected and implemented on a hybrid system that uses MariaDB Columnstore 1.1 for handling columnar data on the DRAM/SSD and SQLite 3.8.3 for executing the row-oriented data processing part on DRAM/SSD. The OLTP performance implies a disk write by the time those transaction log were committed on SSD and the updates were reflected in columnar data on DRAM. Based on that, each transactional/analytical part of the unified system was evaluated by using TPC-C and TPC-H benchmark tests. The OLAP performance was substantially the same, comparing to the original MariaDB Columnstore, but the OLTP performance was increased by approximately 95%.

This study contributes directly for designing serializable replicated systems that supports HTAP workloads, arranging an new MVCC theory on top of global serializability and leading to the implementation policy for the consistent real-time data analysis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Traditionally, most database management systems (DBMSs) have been either write-optimized or read-optimized systems and have focused on utilizing the system architecture for each workload feature. Online transaction processing (OLTP) and online analytical processing (OLAP) applications are categorized as having distinct characteristics, whereby OLTP execution of short-period transactions is required to correctly update small amounts of data and OLAP's long-period ad hoc queries are needed to rapidly aggregate large amounts of data. Accommodating OLTP and OLAP components in a single system has been highly challenging for a long time because there is a performance tradeoff between continuous update propagation and aggregation containing the updated data, which results from differences in workload properties and the type of architecture. Due to the property of connecting OLTP and OLAP components, DBMSs needs to deal with the problem of different access methods for not deteriorating each OLTP and OLAP performances. OLAP accesses data represented in tabular form per column. To enhance the analytical performance, a column-oriented storage model is proposed for efficiently reading the columnar data. The column-oriented storage model can also compress and store the columnar data of same data type in non-volatile storage. OLTP and OLAP processes were originally decoupled and developed by optimizing each workload feature of OLTP or OLAP to improve system performance [Sc18, AMF06, AMH08].

In recent years, hybrid transactional/analytical processing (HTAP) systems have been developed that support both OLTP and OLAP workloads [RCAA20,MGBA17,AKPA17,LMM$^+$13, KN11, NMK15, APM16]. HTAP systems aim to resolve the traditional DBMS problem of unifying OLTP and OLAP, whereby updating and aggregating data can be faster than nighttime batch propagation from an OLTP subsystem operating only in daytime to an OLAP data warehouse system [OTT17]. There is considerable interest in such systems for use in mod-

ern database applications such as real-time data analysis [KPB+19, Pap15, Pla09, doc22]. For example, in [UGA+09], a concrete use case involving the airline industry was investigated.

1.1 Architectures of HTAP systems

HTAP systems established by snapshot isolation (SI) [BBG+07] are categorized as either *unified* or *decoupled* storage architectures, depending on their physical composition [RCAA20]. Systems of using unified storage architecture maintain a single set of snapshot data for transactional and analytical processing shared between compute and storage resources, with isolation between the OLTP and OLAP transaction engines being achieved by SI-based protocols (or "rules"). In unified storage HTAP systems, the OLAP-side transactions can achieve high data-freshness performance, which immediately reads data updated in the OLTP side, but is less successful as a single-node system because of conflicts between the transactional and analytical parts of the co-located storage of the system. Decoupled storage HTAP systems aim to communicate continuously via physically partitioned machines, hoping to achieve better data-freshness performance than nighttime batch propagation without stopping the OLTP subsystem. *Replicated system* comprising both primary and *read-only replica* systems is similar to a decoupled storage HTAP system, but with the read-only replica executing only read-only queries (read-only transactions). An OLAP-side subsystem dedicated to OLAP workloads executes transactions in read-only form because such a read-optimized OLAP subsystem cannot execute update processing efficiently for even a small amount of overwriting. Therefore, the OLTP-side subsystem is used to execute the read-mostly/write transactions.

In addition to the major issues with the performance of HTAP systems, concurrency control (CC) guaranteeing the consistency of updated data has also been an important issue. Transactions under SI cause anomalies in the analysis results, which may contain inconsistent snapshot data [FOO04]. Serializable isolation is contained in higher isolation level than SI, with the serializability guaranteeing that inconsistent database states do not occur. However, aiming to achieve serializability in HTAP systems will lead to reduced performance regardless of the architecture (unified or decoupled architecture.) Performance impact derived from assuring serializability in hybrid of OLTP and OLAP domains is traditional problem. Unified storage HTAP systems can achieve serializability but at the expense of *waits* in or *aborts* of transactions decreasing the OLTP performance. HyPer-Fork [KN11, MKN11] has achieved serializability based on SI [BBG+07] by using the serial execution of OLTP transactions (*writer-wait*) and snapshots obtained from gaps when concurrent update transactions are absent. However, in

seeking serializability for both OLTP and OLAP engines, intentionally creating the gap on the OLTP side would limit the concurrency performance of OLTP write transactions. Therefore, to achieve serializability in OLAP, HyPer-MVOCC [NMK15] aborts OLTP write transactions (*writer-abort*) only when they modify the read sets of OLAP read-only transactions. I consider that an ideal HTAP system should be able to perform OLAP without affecting the OLTP performance.

In a decoupled storage HTAP system, the overall OLTP and OLAP performance takes precedence over achieving serializability. The serializability costs with both OLTP-side and OLAP-side machines are higher than for unified storage architectures because traditional replicated systems tend to use a standard method for supporting serializable isolation, such as bidirectional synchronous communication (2PL, for example). When processing HTAP workloads, the communication overheads in sending back the large read set from the OLAP side to the OLTP side inevitably affects the OLTP performance in decoupled storage HTAP systems. Therefore, decoupled storage systems would worsen the OLTP performance because of the many writer-aborts involved in the OLAP-side scanning of this large read set of data. Consequently, using read-only transactions on a read-optimized OLAP subsystem (or read-only replica) will tend to make the system tolerant toward reading inconsistent data because, when considering the additional tradeoff between performance and consistency, decoupled storage HTAP systems have chosen to enhance performance rather than assuring serializable isolation.

Moreover, in achieving OLAP-side serializability while minimizing the impact on OLTP performance, aborting write transactions on the OLTP-side subsystem would be unrealistic; otherwise it may result in the OLAP-side subsystem having to wait. A replicated system that comprises primary and read-only replicas can produce a technique that HTAP systems will be able to use for CC. For example, the idea for read-only replicas introduced in *SafeSnapshots* [PG12] would provide serializable isolation. Here, transactions executed on the read replica are either forced to wait (*reader-wait*) for the arrival of serializable snapshots that are taken intentionally by writer-abort or to delay the concurrent write transaction starts (writer-wait) in the primary system to obtain successfully serializable snapshots (such as the snapshots taken in the absence of a concurrent writing in HyPer-Fork, or Snapshot Epoch [TZK+13]). If the read-only replica using log-shipping replication (*Streaming Replication* [PG12], etc.) assures serializability without aborting transactions in the primary system, the SafeSnapshots waiting time would be longer because it must then try to obtain a new snapshot via existing read/write (rw) dependencies from outside to inside the snapshots. In the case of workloads generating many rw dependencies, the serializable snapshot waiting time would be longer for

the read-only replica. Therefore, I have considered that the waiting starts in OLTP or OLAP transactions required to assure serializability may also sacrifice too much in terms of OLTP or OLAP performance in cases involving HTAP systems that must remain operational to support real-time data analysis.

1.2 Approach overview

This study focuses on the missing part of achieving serializability in HTAP systems without additional aborts and waits resulting from read-only transaction participation, and my approach prevents read-only transactions from aborting and waiting without stopping the starts of write transactions. This work is aiming initially to achieve serializable isolation between loosely coupled OLTP and OLAP subsystems. Decoupled storage HTAP systems involve the connection of isolated machines and the CC domain is therefore to be considered in the category of distributed transactions. However, the transactions do not run across both OLTP and OLAP machines but can run only in either the OLTP or OLAP machine optimized for the particular workload feature. Therefore, an OLAP-side component such as the architecture creating read-only replicas can regard the federated transaction history as settling the read operations of the read-only transactions into a non-distributed serializable history in terms of conventional transaction theory [WV01,BHG87,SWWW00,BGMS92]. A distributed transaction history can also achieve serializability by certifying that a local history equivalent of the global history is a serializable history derived from the partial-order relations between transactions, even though each subsystem may have a different time-axis from other subsystems. Here, the different time-axes means that no strict time synchronization between machines is required. I expected to have to focus on particular problems (read set transmission, two-phase commit (2PC) and synchronization, for example) on distributed transactions, while also trying to enable the solution to be applied easily to unified storage HTAP systems with reduced aborts and waits in read-only transactions.

This study introduces read-safe snapshots (RSS) as a theoretical framework on top of multiversion CC (MVCC) [ALO00, BHG87] that aims to achieve serializability when performing read-only transaction joins without additional aborts and waits. Using RSSs enables the execution of read-only transactions by selecting serializable and fresh versions of data items. Here, a proposed model assumes that (1) the OLAP-side subsystem executes read-only transactions, (2) the transactions imply that RSSs are applicable, and (3) RSSs are applicable to the OLAP-side subsystem. Read-mostly and write transactions are regarded as OLTP operations.

Precisely because the OLAP-side subsystem is read-only, my proposed method can map the problems of the distributed transaction category into the conventional theory associated with single transaction histories. No matter how many read-only transactions involving RSSs are included, such as a transaction history can be harmless and serializable, even if transactions are executed under the CC domain of decoupled storage HTAP systems. If a replicated system sending RSSs to another machine and reading RSSs is created, it will be easy to construct a decoupled HTAP system by equipping it with serializable read-only replicas. In addition to this proposed approach, I present an RSS construction algorithm that utilizes SSI [CRF09, Cah27] to reduce the RSS construction overhead when tracking transactional dependencies.

This dissertation also aims to address a physical limitation as a particular problem of HTAP systems for enhancing data freshness, OLTP and OLAP performances. To enhance OLAP performance, many HTAP systems have adopted a column-oriented data format on the OLAP-side components, using a traditional way of connection via an extract/transport/load (ETL) process that extracts data from OLTP-side components, transforms data into other formats for analysis, and loads them into OLAP-side components. In decoupled storage HTAP systems that maintain separate storage for transactional and analytical processing, the separation allows replication and optimization of column-oriented data formats [SPP16, BBD+15], together with the preservation and readout of snapshot data in each of the OLTP and OLAP subsystems. Unified storage HTAP systems convert the data OLTP writes on the fly for enhancing OLAP/data-freshness performance. Therefore, regardless of the architecture, HTAP systems must efficiently process the modifications of stored data on OLAP-side components but such systems that combine OLTP and OLAP components cannot execute the data conversion of rows and columns. HTAP systems should be able to perform OLAP with high data-freshness performance without affecting the OLTP performance.

The evolution of hardware devices is remarkable, and DBMSs may be able to overcome performance limits that were limited by physical constraints. DBMSs are middle-layer software that uses the characteristics of devices and applications to optimize performance. In recent years, non-volatile memory (Storage Class Memory: SCM) such as STT-MRAM (spin-transfer torque magnetic RAM) [AKW+13], PRAM (phase-change RAM) [CSP+12], and ReRAM (resistive RAM) [AS10] has been developed for using a persistence device. SCMs are a type of non-volatile RAM with faster access than an SSD. An in-memory HTAP system used SCM for storing transaction logs [ALR+17]. This study propose a DBMS architecture that efficiently uses the SCMs for OLTP and OLAP in real time by creating an update index that updates columnar buffer data on the DRAM and creates a materialized view. Furthermore, the system

uses a mechanism that enable to use more cache domains on DRAM for handling OLAP by performing OLTP using the row-oriented storage method in SCMs. The system was investigated by using TPC-C [Coua] and TPC-H [Coub] benchmark tests for the evaluations of each OLTP and OLAP performance.

With these approaches together, this dissertation describes on the theory and methodology of a high performance serializable HTAP system for consistent real-time data analyses.

1.3 Contributions

The contributions of this study are as follows.

- I develop RSS as a theoretical framework for guaranteeing that a multiversion serializable scheduler can accept read operations targeting recent previous versions when read-only transactions are involved. This scheduler acceptability shows that additional aborts or waits caused by read-only transaction participation are not required to achieve serializability.
  - For a decoupled storage architecture, I show that using RSSs can achieve serializability (abort-free and wait-free) in a read-only replica by constructing the RSS through the log-shipping of collected transactional dependencies in OLTP-side components without sending information (such as the read set) from OLAP-side components and by not using synchronous-commit or 2PL protocols.
- Based on this guarantee, I formalize an algorithm that reduces the cost of tracking transactional dependencies to collect only rw-dependent information for an OLTP engine applying SSI.
- I present the algorithm for two types of HTAP systems: namely, unified and decoupled storage architectures. This shows that my theoretical framework can apply to a wide range of HTAP architectures.
  - I implemented RSS in PostgreSQL and constructed a pseudo decoupled HTAP architecture by using the log-shipping replication function to investigate its effectiveness. I found that, compared with non-serializable isolation in a decoupled storage HTAP system where OLAP transactions were executed under SI on PostgreSQL, the RSS-based decoupled architecture affected both OLTP and OLAP throughput

by only about 15% equipping it with RSSs that ensured serializability. Furthermore, the delay time, reflecting data-freshness performance, for new RSSs to become available was averaged 4.4 seconds, with a maximum delay time was of 10.6 seconds.

  - I confirmed that a unified storage HTAP architecture can achieve serializability on SI-based systems but the OLTP and OLAP performance is degraded by the serializability cost of conventional methods causing aborts or waits in the transactions. In my system, the OLTP throughput was up to 45% higher than that of a hybrid method of SSI on the OLTP side and SafeSnapshots on the OLAP side in a unified storage HTAP system.

- In anticipation of the use of a new device, such as SCM, I proposed an architecture and the method of row/column data conversion. That method is a mechanism that allows a system to preferentially write back column data that tends to be updated, before converting the updated row data to column data for OLAP. This is because the tendency of cached data is different between OLTP and OLAP queries. Consider the data conversion and serializability, in order to determine which versions can be read as serializable and which versions will break serializability when read by OLAP side, that architecture and method assume to temporarily store data on the device that can be accessed at high speed. The proposed architecture presuming SCM was connected by a proposed index and implemented on a hybrid system of MariaDB Columnstore 1.1 for handling columnar data and SQLite 3.8.3 for executing the row-oriented processing part on DRAM/SSD.

  - The system was able to improve the TPC-C result of the evaluation experiment by about 2 times because a large amount of SCM presumed area were vacated by a single swap-out process and the number of blocks to be written back was small.
  - An index for real-time modification of columnar data showed that the processing time of the TiD-based index performs a fast UPDATE query on 25% of the data of scale factor 10 in the TPC-H benchmark relative to the processing time just after performing DBMS X (producted column-oriented DBMS) .

1.4 Organizations

This dissertation consists of seven chapters. Next chapter introduces related literatures. Chapter 3 introduces preliminary knowledge, the problem statement of HTAP systems, and a result of preliminary experiments. Chapter 4 presents the methods of achieving serializable HTAP [1]. Chapter 5 introduces the implementation policy of the HTAP systems [2]. Chapter 6 discusses the future research direction in light of above chapters. The rest chapter concludes the dissertation.

[1] Parts of this chapter are published in [SKA+24]

[2] Parts of this chapter are published in [SHY17]

Chapter 2

Related works

The related works on HTAP systems are reviewed briefly in this chapter. Those methods for efficiently executing both OLTP and OLAP have been traditionally proposed for a long time. Read-optimized systems can be also regarded to contain the transaction processing part of the data transformation from tuple to column.

2.1 SI-based serializability

This section investigates CC in both unified and decoupled storage systems. As mentioned in the Introduction, HTAP systems are categorized as either unified or decoupled storage architectures. Unified storage systems have difficulty in achieving satisfactory OLTP and OLAP performance because of conflicts between transactional and analytical processing when sharing the computing and storage resources. In terms of performance, unified storage HTAP systems also suffer from the aborts and waits required to assure serializability for potentially long-running OLAP read transactions in HTAP workloads. Therefore, decoupled storage HTAP systems that use read-only replicas have been developed in recent years. However, such physically isolated systems turn out to have problems that can restrict the performance when trying to achieve serializability for both of OLTP and OLAP subsystems. As a result, almost all existing HTAP systems that involve executing OLAP under SI have isolated the CC domain from the OLTP subsystem. Unfortunately, SI does not always guarantee serializable behavior because it can lead to inconsistent data such as read-only transaction anomalies [FOO04]. Although previous loosely coupled OLTP/OLAP subsystems (not identified as HTAP systems) have also tried to assure serializability of transactions under SI systems, applying SSI via traditional methods caused aborts or waits and affects the OLTP and OLAP performances. Various optimization

methods have been proposed for assuring serializability of read-only transactions, but have also resulted in aborting or waiting in write or read-only transactions.

Unified storage HTAP system. Although some HTAP systems can support serializability, the OLTP performance can be degraded by the waits or aborts in OLTP transactions caused by assuring the serializability of the OLAP read-only transactions. HyPer [KN11, MKN11, NMK15] achieved SI-based serializability theoretically when involving the participation of OLAP read-only transactions. HyPer-Fork [KN11, MKN11] processed OLTP transactions serially and OLAP queries via consistent snapshots that resulted from copy-on-write for two serial transactions. HyPer-MVOCC [NMK15] avoided write/write (ww) conflicts by locking and determined the visible version of a record by comparing the transaction's start timestamp to the commit timestamp of the version manager in the newest-to-oldest direction. In addition, HyPer-MVOCC executed writer-aborts to prevent rw dependencies [FLO$^+$05] from occurring, thereby achieving serializability under SI. HTAP systems largely utilize snapshot mechanisms for CC and can take advantage of serializability in a similar manner to HyPer. SAP HANA [LMM$^+$13] involved a consistent-view manager that supported SI in the distributed in-memory database. Here, the transactions accessing the consistent-view manager read the most recent committed versions at the time when the transaction began while ensuring the consistent view achieved by SAP HANA. A 2PC protocol ensures the atomicity of distributed multinode update transactions for SI. The write transactions can assure serializability if the 2PC protocol checks for over-write using a lock and aborts any conflicting transaction. Although short-running read-only transactions can reuse a cached consistent view to avoid performance bottlenecks, reading only old data cannot ensure serializability. Long-running read-only transactions can also ensure SI because of the 2PC protocol. Here, the read-only transactions are executed under SI, being derived from consistent view (such as a snapshot). OLAP-side engines may achieve serializable isolation using 2PC and locking methods but executing long-running read-only transactions under SI has been preferred in terms of retaining system performance without degrading OLTP performance.

Decoupled storage HTAP systems. BatchDB [MGBA17] was proposed to isolate based primarily on replication in OLTP and OLAP workloads and on implied SI to realize an HTAP system. Such an architecture could isolate OLTP and OLAP workloads across hardware boundaries, resulting in each replica being generated by a system appropriate for that workload. The OLTP-side CC was based on Hekaton [DFI$^+$13]. Hekaton's MVOCC guaranteed that the transactions could achieve serializability by aborts with re-read validation and ww conflict detection [LBD$^+$11]. Considering the performance overheads of serializability, the OLTP

component in BatchDB was not used in Hekaton's serializable isolation level. In contrast, the OLAP component did not assure serializability but achieved SI in the replicated system and maintained single-version data individually copied from one batch. To maintain consistent views, the OLAP replica executed transactions nonconcurrently when executing the batch propagation. Even though OLAP queries read consistent snapshots, which means no concurrent-overwrite in the OLAP-side components, transactions would generate a nonserializable history as long as the OLTP side keeps running (i.e., overwriting). If the OLAP-side transactions had read a very old version and the OLTP side had nonconcurrently created the immediate successor version, this must be regarded as rw antidependency (or called rw conflicts [WV01, BHG87, SWWW00, BGMS92].) Such dependencies derived from transactions under SI would lead to a cycle in a dependency graph [ALO00], in addition the transactions may non-serializable history by read-only transaction participation [FOO04]. Therefore, decoupled systems that involve read-only replicas have tried always to send and check read sets for each read-only transaction to the master machine because of the serializability costs associated with overwrite transaction aborts.

There are several ways of using SI to enable serializable isolation. SSI [CRF09, Cah27] is known as a theory offering serializability under SI. The nonserializable state of SI transactions contains two successive concurrent rw dependencies called a "dangerous structure" [FLO$^+$05]. If a system that uses SI aborts one of the transactions containing two successive rw antidependencies, the system can achieve serializability. However, applying SSI to HTAP systems leads to the OLAP performance worsening because of reader-aborts. Because OLAP's read-only transactions tend to involve very large read sets and have long lifetimes based on the characteristics of scan-mostly and long-running analytical queries, detections of concurrent rw dependencies could be time-consuming. RSSI [JHFR11] is an SSI implementation for a distributed environment. Here, the transactions on each machine have to send their read sets to each other to detect rw dependencies and dangerous structures. In particular, recent HTAP systems using replication need to send a transaction's read sets from the read replica dedicated to the OLAP workload to the OLTP engine, and write transactions concurrent with long-running read transactions must keep track of (over-)writes. If two short-running update transactions are committed as the first rw dependency before a read-only transaction creates a second rw dependency, a reader-abort of the second rw dependency would be executed. In addition, if a long-running read-only transaction is retried by a serializability certifier, a reader-abort could easily occur again.

CP-ROO [BHEF11] is an SI-based read-only optimization of the RSSI system. CP-ROO

can improve the OLAP performance but the OLTP performance may degrade when applied to HTAP systems, because its validation chooses writer-abort instead of reader-abort when it detects rw dependencies. Prefix-consistent snapshot isolation (PCSI) [EPZ05] supports serializability on SI-based protocols for distributed systems. PCSI selects reader-abort instead of writer-abort on the transactions having a read–write relation. CockroachDB [TSM+20] is a high-performance system that also tries to achieve serializability via an implementation approach involving timestamp management in the SI read protocol and write transactions that write provisionally until commit. However, the read-only (non-leaseholder) replicas must reuse the validation results of write transactions and read the validated no overwritten data. The problem of assuring no future writes was solved by writer-wait rather than writer-abort (using provisional writes) and the certification transactions cannot overwrite if there has already been a read. However, serial execution (writer-wait) or writer-abort may worsen OLTP performance in cases where the systems processes HTAP workloads.

For the read-only replicas in log-shipping replication, Ports and Grittner [PG12] proposed taking snapshots (SafeSnapshots) for serializable isolation based on SI. Because read-only transactions under SSI can read a snapshot without the need to track rw dependencies, SafeSnapshots were able to be exploited in a system that offered SSI. The discussion of of *Streaming Replication* described in [PG12] illustrated the idea of taking reliably safe snapshots the read-only replicas in log-shipping replication, while using writer-abort or writer-wait in the OLTP engine and reader-wait in the read-only replica. That is, such transactions, called "deferrable transactions" in this dissertation, must abort concurrent write transactions or must prevent new transactions from starting while waiting for concurrent write transactions to finish. Moreover, read-only transactions in the replica must wait for SafeSnapshots to arrive, which is why the idea was not implemented for the read-only replicas (it would degrade the OLTP or OLAP performances). In unified storage architectures, the system can create SafeSnapshots for optimization of the read-only transactions.

2.2 Column-oriented storage model

DBMS has traditionally been used to execute OLTP, even before HTAP was mentioned. In order to handle OLTP efficiently, DBMSs generally adopt a row-oriented storage model. DBMS generally accesses data in blocks of a storage, which is known as block-oriented query processor [PMJA01]. This block can contain tens or hundreds of tuples, so it's a middle ground between processing tuples at once and materializing a full table. Performance generally im-

proves with increasing block size, as long as the cumulative size of the tuple blocks flowing between operators in the query plan fits in the CPU cache. A major advantage of block-oriented processing is the reduction in the amount of method calls [ZNB08]. Column-oriented storage model came to be used due to the increased demand for data analysis for use in decision making [Bay98]. Column-oriented DBMS is able to significantly improve the execution speed of OLAP [AMF06]. This occurs because the DBMS optimizes the block-oriented processing and the storage model based on the unit of the column for performing OLAP. Column-oriented storage model is only different in that it stores data to disk by units of columns from a table. The rest of this section describes the processing methods that are adopted for column-oriented DBMS with OLAP and the storage methods in OLAP.

Column-oriented DBMSs sequentially store columns of table data in disk pages. Thereafter, it can efficiently compress the column by sorting the same type of data because of the similarity of data in the page. C-store and MonetDB have been proposed for open source on a column-oriented system, and they are being developed as commercial systems [LFV+12, RBZ13]. Compressed and sorted for storage on a disk, the column can have several compression methods applied, such as run length encoding and bit-vector encoding, based on the features or cardinality of the data in columns [WOS02]. Moreover, the DBMSs use a modern CPU to load the compressed data into its SIMD register; therefore, the columnar systems can take advantage of processing the same type of data in columns and more overall data based on compression [RBZ13, AMF06]. For compression efficiency, the OLTP data executed in units of rows is stored on DRAM as segments (or row groups), which are the units of data stored on disk, and the maximum and minimum values in the segment on DRAM are stored for each column. A method has also been proposed in which the segments on the disk corresponding to the interval are read out, and the multiple segments are sorted and stored on the disk [SPP16]. By sorting segments together, in addition to efficiently skipping segments when reading data on the disk, it was possible to read consecutive segments together efficiently. RCFile, ORCFile and Parquet [HLH+11, Apab, Apaa] is also a column-oriented file format that can be easily used in distributed storage for large-scale data processing such as MapReduce. They distribute data by replicating data to buddy nodes in units of projections by creating a table structure called a projection in units of multiple columns from the attribute columns of the original table, as in Vertica [LFV+12]. If row data tuples are distributed among nodes when distributing data, it is necessary to reconstruct the distributed tuples across the network, which is likely to become a bottleneck in MapReduce, etc. Therefore, those column-oriented file format for distribution adopts a method different from column groups and column stores by sorting and storing tuples

by row group corresponding to the column-oriented DBMS segment in one node without dividing the tuple. are doing. By adopting a method of compressing each column in a row group, it is possible to read the row group header first and skip to the next row group if the query WHERE condition is not met.

However, column-oriented storage has a disadvantage when processing OLTP. In the case of processing many columns of a table, row-oriented storage is better than column-oriented storage because columns divided from a table require a row to be reconstructed on multiple columns. For this inefficient case, therefore, a materialization called projection in Vertica [LFV+12] separates the table into several columns based on certain sort patterns and stores the column groups to the disk [AMDM07]. The projection is able to read a row of the columns from the disk using compressions and is more efficient than multiple columns that use row-oriented storage such as PAX [ADHS01]. The column groups on the disk can take advantage of the row-oriented storage by physically storing the related columns in the same disk page [PAZN05]. In addition, the column grouping pattern for efficient OLAP is based on several entities and sort patterns because users that handle the column-oriented DBMS for DWH tend to perform queries corresponding to the patterns. Column-oriented storage is not efficient for performing OLTP because OLTP handles data by a tuples. Therefore, once the column-oriented DBMS performs OLTP on row-oriented storage on the DRAM, it stores the sorted and compressed columns on the disk when the data accumulating on the DRAM reached a segment of tuples [SAB+05].

A storage method on the RAM called delta buffer controls the OLTP of INSERT and DELETE operations by adopting value-based delta tree (VDT) in Vertica [LFV+12]. VDT uses a tree structure for handling sorted data on the disk and the RAM, and refers to INSERT tables and DELETE tables on the RAM. These tables are stored on the disk using a merge union method and a merge diff method. The merge union method merges the sorted data based on the order of INSERT tables on the disk and on the RAM. On the other hand, the merge diff method stores the sorted on-disk data that does not exist in the DELETE tables on the RAM. Moreover, the PDT (positional deleta tree) extends the VDT for possessing the smaller sized delta buffer [HZN+10]. PDT can perform UPDATE in the delta buffer, unlike UPDATE in VDT, which is divided into INSERT and DELETE. Column-oriented DBMS has a sort key based on values of specific columns such as the projection; therefore, the delta buffer on the RAM must read the sort key from the disk to store the sort keys in real time using OLTP. The PDT performs OLTP by managing the tree structure similar to a B-tree based on SID as the sort key in the disk; it updates the sort key as RID on the delta buffer. OLTP is carried out using

memory INSERT to an insert table, using DELETE to a delete table that stores only the sort key values defined by the table on the disk, and using UPDATE to an array corresponding to modifying the columns in the table. PDT can calculate RID without storing it as data because it stores the sort key, which reads the SID of the top node as a pivot of the hierarchy structure, and the tree structure stores the number of INSERT - DELETE operations as delta in the each node that can define the smallest RID of the right side nodes.

In addition, column-oriented storage that uses the delta buffer tends to store the column data by splitting it into units called segments, such as several hundred thousand tuples for reading sparse data from the disk. MemStore [SPP16] adopts metadata of the minimum value and the maximum value in the segment for keeping sorted data in the segment or between the current segment and the next segment. Because the delta buffer stores a segment to the disk in a block, the method efficiently performs OLAP by reading data sorted between the segments on the RAM and segments on the disk. The segment sorting method prepares the number of segments that have the minimum value of a segment on the disk, rather than the maximum value of the segment on the RAM, and have the maximum value of a segment on the disk, rather than the minimum value of the segment on the RAM. The method stores to disk using sorting data from the prepared segments. To efficiently perform OLAP and OLTP in column-oriented DBMSs, which requires many reads of data from the disk, the sparse reading method (called segment elimination) conducts efficient column-cased disk I/O using the metadata of minimum and maximum values. Moreover, tuning methods that use OLAP queries are also efficient for sparse reading because the users of the column-oriented DBMSs perform similar queries based on the considered features of the data [ŚE09].

Chapter 3

Preliminaries

This chapter prepares for proposal's formal representation and organizes the underlying models and theories [WV01, BHG87, SW00, SWWW00, BGMS92, ALO00, CRF09]. Section 3.1.2 describe the assumptions related to distributed transactions in a decoupled storage HTAP system, where there is one OLTP side and one OLAP side and each transaction involves just one side.

3.1 Database model

This research uses a multiversion-history formalization [ALO00] and SI definitions [SW00]. The database comprises data objects (or data items) that can be read or written by transactions. Each row or tuple is a data item, which have one or more versions. A history H over a series of transactions has two parts: namely, a partial order on the operations (e.g., read, write, commit, abort) of those transactions; and a version order on the versions of data items written by committed transactions. Here, with reference to history H and operations such as $o_i, o_j \in H$, I denote $o_i \prec_H o_j$, if o_i precedes o_j in H. Let $\mathbb{T} = \{T_1, \ldots, T_n\}$ be a set of transactions. A transaction $T \in \mathbb{T}$ is a (totally ordered) sequence of read and write operations on data items, together with a begin and a commit operation such that a begin operation precedes all read and write operations in T and a commit operation follows all other operations in T. It preserves the order of all operations within a transaction including the commit and abort operations. For transaction T_a, I write $Begin(T_a)$, or $B(T_a)$, as an operation that appears before the earliest preceded operation in T_a's operations (within a history). Similarly, I write $End(T_a)$, or $E(T_a)$, as a operation that appears after the last successor operation in T_a's operations in a history if T_a's execution is successful. When a transaction writes a data item X, it creates a new version of X. A write operation on a data item X via transaction T_a is denoted by $W_a(X_a)$. Data item

X_a is T_a's final-modification version with respect to X. If it is useful to indicate the value v being written into X, I use the notation, $W_a(X_a, v)$. I describe $\ll$ as a version order, which is the total order on versions of a data item written (or installed) by committed transactions in H (I refer to versions generated by committed transactions as committed versions). For example, if transaction T_b wrote X and committed after X_a was written by committed transaction T_a, I would write $X_a \ll X_b$. When transaction T_a reads a version of X that was created by T_b, I denote this as $R_a(X_b)$. If it is useful to indicate the value v being read, I use the notation $R_a(X_b, v)$. If an operation $R_b(X_i)$ exists in a history, it is preceded by $W_i(X_i)$ in the history: transaction T_j cannot read version X_i before it has been written by T_i.

I denote a transaction T_a that executes in the OLTP-side subsystem by $T_a^{(t)}$. Similarly, I denote a transaction T_a that executes in the OLAP-side subsystem by $T_a^{(a)}$. I also describe $R_a^{(a)}(X_b)$ as a read operation if OLAP transaction $T_a^{(a)}$ reads a data version X_b that resides in the OLAP-side subsystem and is written by the OLTP committed transaction $T_b^{(t)}$. If OLTP transaction $T_a^{(t)}$ reads a data version X_b written by $T_b^{(t)}$ in OLTP-side subsystem, I write $R_a^{(t)}(X_b)$ as a read operation. I denote a write operation in transaction $T_a^{(t)}$ on version X_a of data item that resides in the OLTP-side subsystem by $W_a^{(t)}(X_a)$.

Although the version read by T_a is not necessarily the most recently written and committed version, read operations under SI specify the versions written by the most recently committed transactions before T_a began. Schenkel and Weikum [SW00] have defined SI as a multiversion history meeting the following two conditions. First, the SI version function (SI-V) is the read protocol that maps each read operation $R_a(X)$ to the most-recent committed write operation $W_b(X)$ as of the time $B(T_a)$ for the begin operation of T_a. Second, the SI write function (SI-W) is a commit or write protocol (called "first committer/updater wins" rule) for which the write sets of two concurrent transactions are disjoint. Read/write/commit operations of a transaction under SI are based on these protocols (or rules).

3.1.1 Serializability

Adya et al. [ALO00] have defined the direct serialization graph (DSG) arising from any given multiversion history, as follows. Each node in DSG(H) corresponds to a committed transaction, which is a transaction performing an end operation in a history H, and directed edges correspond to three types of *direct conflict*, as follows. For committed transactions T_a, T_b and data item X on history H under SI, I say that first, a ww conflict from T_a to T_b occurs if T_a writes X_a and T_b writes X's immediate successor version in the version order; second, a write/read (wr) conflict from T_a to T_b occurs if T_a writes X_a and T_b reads X_a; and third, a rw

conflict from T_a to T_b occurs if T_a reads data-item version X_i and T_b writes X's immediate successor version in the version order. I also define a $T_a \rightarrow T_b$ dependency edge (directed dependency edge) as a directed edge in the DSG if the directed edge corresponds to any of the existing direct conflicts (ww, wr, or rw conflicts) from T_a to T_b. If DSG(H) is acyclic, H guarantees serializability, in which case H is called a "serializable history".

I say that T_b is *reachable* from T_a if there is a path of dependency edges from T_a to T_b that can be reached directly or via other transactions without making a cycle within DSG(H). I denote this as a $T_a \rightarrow^* T_b$ reachable dependency edge if T_b is reachable from T_a. Reachable dependency edge $T_a \rightarrow^* T_b$ represents a transitive dependency edge starting at T_a and ending at T_b without including reversed direction dependency edges from T_b to T_a, and $T_a \rightarrow^* T_b$ implies reachability in a digraph DSG given a serializable history. Transactions satisfying serializability cannot construct the cycles on DSG. For example, for a DSG from given serializable history $H(T_a, T_b, T_c)$ restricted over transaction set $\{T_a, T_b, T_c\}$ satisfying serializability of Adya et al. [ALO00], if $T_a \rightarrow T_c$, $T_c \rightarrow T_b$ exists and a reversed-direction dependency edge from T_a to T_b does not exist (none of $T_c \rightarrow T_a$, $T_b \rightarrow T_c$, or $T_b \rightarrow T_a$), I can specify $T_a \rightarrow^* T_b$. Note that $T_a \rightarrow^* T_a$ always holds for one transaction T but it is not considered to be a cycle. I also specify a $T_a \not\rightarrow^* T_b$ unreachable dependency if there is no path of dependency edges reachable from T_a to T_b directly or via another transaction in DSG(H). I say that T_b is *unreachable* from T_a if $T_a \not\rightarrow^* T_b$ holds.

Adya et al. introduced the isolation level PL-3 [ALO00], which ensures serializability within the class of multiversion conflict serializabilities [WV01]. PL-3, as applied to a multiversion-history H, proscribes the following three situations: a situation where a committed transaction reads a version of a data item written by an aborted transaction; a situation where transaction T_a reads a version of data item X written by transaction T_b that was not T_b's final modification of X; and a situation where DSG(H) contains cycles with at least one rw dependency edge. A multiversion history generated under SI would proscribe the first and second of these situations.

PL-3 more widely covers a serializable history under SSI. There are methods of ensuring serializability based on SI. Cahill et. al. [CRF09] proposed SSI, which assures serializability of the histories based on SI. When transactions executed under SI creates two successive rw dependencies (i.e., "dangerous structure") [FLO$^+$05], SSI rejects the history by aborting a transaction within the dangerous structure instead of finding complete cycles in the DSG, thereby assuring serializability of the history under SI. SSI can accept a wider history range of histories beyond those involving 2PL [BHG87, WV01]. The 2PL of a history under SI cannot

accept overwrites creating a single rw dependency, which assures serializability under SI, but SSI can accept the history up to the creation of a single rw dependency. Therefore, SSI can accept the operations in a concurrent update transaction. A history generated under PL-3 can accept operations of transactions under SI untill complete cycles in the DSG are created. Therefore, the serializability class of a SSI history is smaller than the scheduler-acceptability class of PL-3. PL-3 can contain a serializable history for which SSI would be generated, because the version order is equivalent to the order of the commit operations by SI-W [SW00]. However, SSI can avoid increasing the cost of finding all dependencies (ww, wr, and rw dependencies) by limiting the search to rw dependencies alone.

3.1.2 HTAP scheduler

In this study, I adopt the multiversion-history formalization in [ALO00] and assume that a multiversion history (also called the "schedule") is always given with a version order of writes. I call the class of the multiversion history of full schedule satisfying PL-3 serializability *version-ordered conflict serializability (VOCSR)*. In other words, VOCSR is the set of multiversion histories with a given version order of writes that are serializable. I use the definition of history described in previous Section 3.1. A transaction cannot read a version before it has been written by other transaction. Moreover, the order of ww, wr, and rw conflicts derived from read/write operations in a history of partial order corresponds to dependency edges on digraph DSG. The computation of serializability in VOCSR is required to verify that the graph is acyclic.

An *HTAP scheduler* processes operations in transactions dynamically (non-deterministic scheduler), which implies isolation of the OLTP and OLAP protocols from each other. An example is a system configuration with an OLTP-side subsystem supporting SSI and an OLAP-side subsystem supporting SI. An *HTAP history* is a sequence formed from the union of the operations from given transactions in each OLTP and OLAP. I aim to design an HTAP history acceptable to a HTAP scheduler that achieves spontaneous serializability of combinations of OLTP and OLAP operations. The OLAP-side subsystem assumes that read operations can know the operations of transactions from the OLTP-side subsystem, except for write operations yet to be committed at the time of decoding information from the OLTP side. The HTAP scheduler presupposes the following conditions.

- the OLTP transaction does not perceive the processing contents of OLAP. When appending an OLTP operation, the OLTP protocol knows only the OLTP operations in the history prefix.

- An OLAP transaction can perceive the processing results of OLTP (through log-shipping or other processes). When appending an OLAP operation, OLAP protocols know the operations of transactions via the history prefix, which includes the version order derived from SI-W on OLTP side.

The HTAP scheduler represents the history for a decoupled storage HTAP system. I assume that a decoupled storage HTAP system contains primary and read-only replicas to enable a focus on serializability. Although for a decoupled storage HTAP system it is possible that OLTP and OLAP operations are executed in a distributed manner (such as for a system configured with multiple OLTP subsystems and multiple OLAP subsystems), I will analyze only systems containing one of each type. The OLAP-side subsystem can assume that it is not efficient to execute the update or delete processing results from the OLTP for even a small amount of overwriting because the system aims to optimize the use of compute and storage resources with respect to the efficient processing of the OLAP workload [Sc18, AMF06, AMH08, RCAA20, MGBA17]. I assume that transactions in the decoupled storage HTAP system do not involve both OLTP and OLAP subsystems, but run only in the one appropriate subsystem for processing (either the OLTP or the OLAP subsystem). In this research, a history of OLTP-side components (or subsystems) assumes an SI-based serializable history in the class of VOCSR as shown in Figure 3.1. Therefore, the read operations in OLTP transactions read data versions based on SI-V and the version order are determined by SI-W, but read operations in OLAP read-only transactions map data versions without being restricted by SI-V. The OLAP side can perceive the version order derived from OLTP side commit order via log-shipping from OLTP-side subsystem. I aim to formalize the read protocol to ensure serializability.

The serializability of a decoupled storage HTAP system is computed via the synthesis of the two histories, in the OLTP subsystem and in the OLAP subsystem. To naïvely achieve serializability, bidirectional communication between the OLTP and OLAP subsystems would seem essential because of having to collect information from both subsystems to compute the serializable transaction order for the partial-order relations. However, bidirectional communication becomes a problem because ensuring serializability worsens the hybrid system performance. I illustrate the problem and my approach via Figure 3.3 in the next section.

3.1.3 The missing piece: read-only anomalies

Because performance degradation can be large in decoupled storage HTAP systems, bidirectional synchronization communication is not purposely preferred for the OLAP read-only repli-

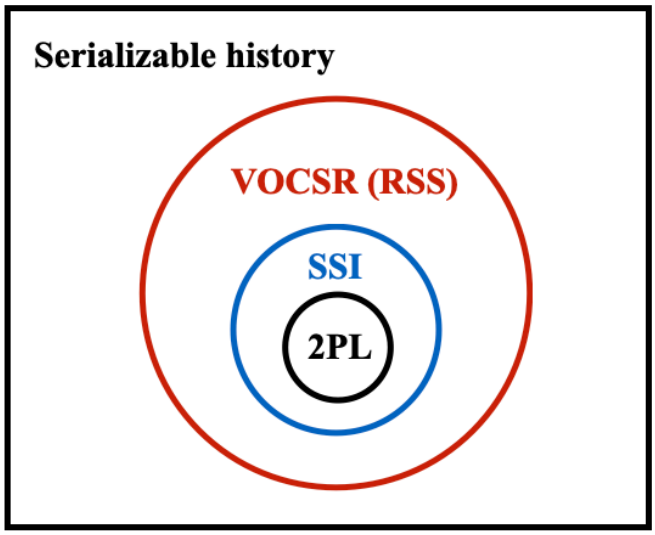

Figure 3.1: Inclusion relation of multiversion-conflict serializable histories on scheduler acceptability.

cas, but the result is that anomalies can occur. An anomaly can occur when additional OLAP read-only transactions participate in the serializable history but are only verified by the OLTP-side subsystem. The situation causing anomalies has to be considered when designing a system configuration that includes unidirectional communication from the OLTP-side subsystem to OLAP-side subsystem. I should identify the anomaly situations for read-only transactions in a snapshot received from OLTP-side subsystem in decoupled storage HTAP systems.

Definition 1 (Read-only anomaly in an HTAP schedule) *I say that a read-only anomaly occurs if a history H over transactions $\mathbb{T}$ contains a nonserializable structure $\mathbb{S} \subseteq \mathbb{T}$, where a set of transactions $\mathbb{S} = \{S_1,\ S_2,\ S_3,\ \cdots,\ S_n\}$ meets the following conditions.*

- *History H, restricted over transactions $\{S_1,\ \cdots,\ S_{n-1}\}$, is a serializable history in VOCSR:*

  $H(S_1,\ \cdots,\ S_{n-1}) \subseteq VOCSR$,

- *S_n is a read-only transaction, and*

- *$\mathbb{S}$ has a cycle of dependency edges P;*

  $P = \{S_1 \rightarrow^* S_2,\ S_2 \rightarrow^* S_3,\ \cdots,\ S_{n-2} \rightarrow^* S_{n-1},\ S_{n-1} \rightarrow S_n,\ S_n \rightarrow S_1\}$. *Note that the last two edges $S_{n-1} \rightarrow S_n$ and $S_n \rightarrow S_1$ are a direct dependency.*

In the following example, I have a case of a read-only anomaly occurring in an HTAP history, where I denote the commit operation of OLTP transaction $T_a^{(t)}$ by $C_a^{(t)}$ and denote the commit operation of OLAP transaction $T_b^{(a)}$ by $C_b^{(a)}$. The HTAP scheduler that generates the HTAP history assumes that two conditions apply: namely, that OLTP-side components adopt SSI via a concrete VOCSR protocol and OLAP-side components adopt SI. The OLAP-side subsystem knows about the operations in OLTP transactions by decoding the information (such as WAL) sent from the OLTP-side subsystem. The order of transactions can be determined for read and write operations even though the HTAP subsystems are decoupled.

Example 1 (Read-only anomaly in an HTAP schedule) *I describe an example demonstrated by Fekete et al. [FOO04], where I assume that OLTP transactions are executed under SSI (as a form of VOCSR) and OLAP transactions are executed under SI. Suppose that X and Y are two data items representing checking and savings account balances, respectively. Assume $X_0 = 0$ and $Y_0 = 0$ as initial conditions. In the following history, OLTP transaction $T_1^{(t)}$ deposits 20 to the savings account Y, $T_2^{(t)}$ subtracts 10 from the checking account X, considering the withdrawal covered, provided $X + Y > 0$, but accepting an overdraft with a charge of 1 if $X + Y$ goes negative, and $T_3^{(a)}$ is an OLAP read-only transaction that retrieves the values of X and Y and prints them out for the customer. In one sequence of operations these transactional operations will result in the following history:*

$$H_{roa} : R_2^{(t)}(X_0, 0)\ R_2^{(t)}(Y_0, 0)\ R_1^{(t)}(Y_0, 0)\ W_1^{(t)}(Y_1, 20)$$
$$C_1^{(t)}\ R_3^{(a)}(X_0, 0)\ R_3^{(a)}(Y_1, 20)\ C_3^{(a)}\ W_2^{(t)}(X_2, -11)\ C_2^{(t)}.$$

The anomaly that arises in this transaction is that the read-only transaction $T_3^{(a)}$ prints out $X = 0$ and $Y = 20$. This could not happen in any serializable execution that ends in the same final state as H_{roa}, with $X = -11$ and $Y = 20$, because if 20 were added to Y before 10 was subtracted from X in any serial execution, no charge of 1 would ever occur, and the final balance should then be -10, not -11. The customer, knowing that a deposit of 20 was due and worried that his check for 10 might have caused an overdraft charge, would conclude that he was safe, based on this balance result. Indeed, such a print-out by transaction $T_3^{(a)}$ would be very embarrassing for the bank itself should bank regulators ask how the charge was incurred.

The SSI of the OLTP side cannot identify the dangerous structure $T_3^{(a)} \rightarrow T_2^{(t)} \rightarrow T_1^{(t)}$ derived from the OLAP read-only transaction $T_3^{(a)}$ in HTAP history H_{roa}. The read operations of $T_3^{(a)}$ perceive the operations in the history prefix up to $C_1^{(t)}$ via the decoding log sent from

the OLTP-side subsystem, by which $T_3^{(a)}$ under SI-V reads the most-recent-committed version of data items Y_1 and X_0 in their version order $[Y_0 \ll Y_1,\ X_0 \ll X_2]$. Direct dependency edge $T_2^{(a)} \rightarrow T_1^{(t)}$ occurs by committing transaction $T_2^{(t)}$ in the OLTP-side subsystem, where SSI accepts the operations of $T_2^{(t)}$ because the dangerous structure has yet to be created. $T_3^{(a)} \rightarrow T_2^{(t)}$ occurs by accepting operation $R_3^{(a)}(X_0)$ reads the previous version of X_2 before $C_2^{(t)}$ arrives in the OLAP-side subsystem, because the version order $X_0 \ll X_2$ is decided in the OLTP-side subsystem. Moreover, $T_1^{(t)} \rightarrow T_3^{(a)}$ occurs by $R_3^{(a)}(Y_1)$, as written by $T_1^{(t)}$, because $T_3^{(a)}$ identifies Y_1 as the most-recent version in Y's version order $Y_0 \ll Y_1$ from the order of commit operation $C_1^{(t)}$ derived from SI-W (T_0 implies the initial state of the database and the operations of T_0 are omitted). H_{roa} does not achieve serializability, because DSG(H_{roa}) contains the cycle $T_3^{(a)} \rightarrow T_2^{(t)} \rightarrow T_1^{(t)} \rightarrow T_3^{(a)}$, which implies that serializability cannot be achieved. Read-only anomalies occur when naïvely reading consistent snapshots (SI) in the OLAP subsystem, even though the OLTP-side subsystem supports VOCSR obtained from verifying a complete cycle.

I also confirmed that a read-only anomaly occurs in the CH-benCHmark [CFG+11], which is a benchmark test that is actually used in the evaluation of HTAP systems. The CH-benCHmark has two parts, derived from TPC-C [Coua] and TPC-H [Coub], which issue OLTP and OLAP transactions, respectively. There are OLTP transactions issued in CH-benCHmark, named "Delivery" and "NewOrder", in the history H_{roa} described above, where the Delivery transaction corresponds to $T_1^{(t)}$ and NewOrder transaction corresponds to $T_2^{(t)}$. The relevant operations in H_{roa} correspond to those in CH-benCHmark as follow. $R_2^{(t)}(Y_0)$ is a read operation in the NewOrder transaction for updating data in a table named "Customer", $R_2^{(t)}(X_0)$ is a read operation (and $W_2^{(t)}(X_2)$ is a write operation) in the NewOrder transaction for updating data in a table named "Stock", and $R_1^{(t)}(Y_0)$ and $W_1^{(t)}(Y_1)$ are read and write operations in the Delivery transaction for updating data in the Customer table. $T_1^{(t)}$ and $T_2^{(t)}$ can be executed in parallel; i.e, they do not succeed if the OLTP-side subsystem adopts 2PL but would be allowed in SSI, thereby creating the dependency edge $T_2^{(t)} \rightarrow T_1^{(t)}$. In addition to this example, the Delivery transaction corresponding to $T_1^{(t)}$ creates a new data version to update data items in the tables called "Orders" and "Order-Line".

CH-benCHmark issues several OLAP transactions that read the data that $T_1^{(t)}$ and $T_2^{(t)}$ have generated, among them queries labeled Q5, Q7, Q8, Q9, and Q21, each of which corresponds to $T_3^{(a)}$. The queries read the following: data created by $T_1^{(t)}$ and $T_2^{(t)}$, the data in the Order and Order-Line table created by $T_1^{(t)}$ corresponding to $R_3^{(a)}(Y_1)$, and the data in the Stock table cre-

ated by $T_2^{(t)}$ (correspond to $R_3^{(a)}(X_0)$ reading the previous version of X_2). Dependency edges $T_1^{(t)} \rightarrow T_3^{(a)}$ and $T_3^{(a)} \rightarrow T_2^{(t)}$ exist between the relationships derived from the operations, creating the cycle $T_3^{(a)} \rightarrow T_2^{(t)} \rightarrow T_1^{(t)} \rightarrow T_3^{(a)}$. Depending on the read data, the dependency edge will change either to wr dependency or rw dependency, but because it is common to evaluate system performance by issuing transactions simultaneously without controlling the order of execution, it is conceivable that read-only anomalies could occur. Although this example is not as easy to understand as the previous example of bank transactions, I believe that anomalies should always be prevented so that fatal problems do not occur later because various new applications and user demands begin to be used.

In what follows, a preliminary experiment reports the cycles of read-only anomalies to occur in CH-benCHmark. The detailed implementation and setup of the preliminary experiment are described in Section 4.2 and Section 4.3.2. Those sections display the implementation and the experimental way on my multinode system such as decoupled storage HTAP systems. The proposed system assumed that the transactions in the OLTP-side subsystem are executed by serializable isolation but the OLAP-side subsystem is set by snapshot isolation to detect read-only anomaly cycles. Because those OLAP queries issued by CH-BenCHmark do not basically abort due to the execution of SI. The OLTP-side serializable method is based on SSI, thereby causing the read-only anomaly structures (cycles) such Definition 1. The experiment set Scale Factor (SF) to 10 and execute OLTP and OLAP parallel clients 16, 24, and 32, respectively, and ran one experiment for 100 seconds. Cycle occurrence was derived from Definition1 and the ratio computed by only unique OLAP query contained in the read-only anomaly cycles because it can regard the cycle structure as transforming serializable history by executing one-abort-at-a-time against an OLAP query. Table 3.1 summarize the average of the five runs.

Table 3.1: Occurrence ratio of read-only anomaly cycles in CH-BenCHmark (SF=10) under a decoupled storage HTAP system.

Measurement items	OLAP clients	OLTP clients 16	24	32
Cycle occurrence rate (%)	16	9.3	12.1	13.6
	24	7.9	10.5	12.3
	32	6.6	10.7	12.5
OLTP throughput (TPS, transaction per second)	16	3422.1	4329.1	4760.4
	24	3366.5	4334.7	4686.7
	32	3344.7	4148.7	4514.9
OLAP throughput (QPH, query per hour)	16	11270.0	10917.3	10890.0
	24	11569.3	11881.0	11781.8
	32	11815.5	11577.5	11419.1

Although cycles had been caused in the CH-BenCHmark, I found that the OLAP queries in CH-BenCHmark had not read the attributes modified by OLTP transactions, therefore, anomaly values would appear in the layer of users or application side. In Table 3.2, an evaluation additional reports the user-level anomaly on the experiments with using an additional OLTP and OLAP-side transaction. In the experiments, Credit Check transaction of TPC-C++ [ALO00] was add in the OLTP side of CH-BenCHmark, causing data written by the transaction were read by newly adding as an OLAP-side query of #23 in CH-BenCHmark;

```
[Credit Check transaction]

SELECT c_balance, c_credit_lim
INTO c_balance, c_credit_lim
FROM Customer
WHERE c_id = c_id
AND c_d_id = d_id  AND c_w_id = w_id

SELECT SUM(ol_amount)
INTO neworder_balance
FROM OrderLine, Orders, NewOrder
WHERE ol_o_id = o_id AND ol_d_id = d_id AND ol_w_id = w_id
AND o_d_id = d_id AND o_w_id = w_id AND o_c_id = c_id
```

```
AND no_o_id = o_id AND no_d_id = d_id AND no_w_id = w_id

if (c_balance + neworder_balance > c_credit_lim)
        c_credit = "BadCredit";
else
        c_credit = "GoodCredit";

UPDATE Customer
SET c_credit = c_credit
WHERE c_id = c_id AND c_d_id = d_id AND c_w_id = w_id;
COMMIT;
   (Credit Check transaction contained in the OLTP side of CH-BenCHmark)
```

```
[ #23 of OLAP query ]
SELECT o_w_id, o_d_id, o_c_id, no_balance, c_balance, cs_credit
FROM credit_status, customer,
     (SELECT o_w_id, o_d_id, o_c_id,
             SUM(ol_amount) AS no_balance
                FROM order_line,
                        (SELECT o_w_id, o_d_id, o_c_id, o_id
                         FROM o_order
                         INNER JOIN new_order
                        ON o_w_id = no_w_id
                        AND o_d_id = no_d_id
                        AND o_id = no_o_id) AS tmp
                WHERE ol_w_id = o_w_id
                        AND ol_d_id = o_d_id  AND ol_o_id = o_id
                GROUP BY o_w_id, o_d_id, o_id, o_c_id) AS no_bal
WHERE no_bal.o_w_id = cs_w_id
  AND no_bal.o_d_id = cs_d_id  AND no_bal.o_c_id = cs_c_id
  AND cs_w_id = c_w_id  AND cs_d_id = c_d_id
  AND cs_c_id = c_id;
   (OLAP side executed only this OLAP query)
```

These transactions change the value depending on the balance and read the updated value on the OLAP side, so when the user actually prints it out on paper etc., it will give the customer the wrong explanation. Credit Check transaction was set to occur with a probability of 50% and the OLAP-side subsystem executed only the added query as an OLAP query of #23. The experiment was run for 60 seconds with SF set to 4. The result is the rate of occurrence of read-only anomaly cycles that actually cause user-perceived anomalies. This study was aimed not at whether the application-layer anomaly actually occurs but at the cycle itself not being

created (to assure serializability).

Table 3.2: Occurrence ratio of user-level read-only anomalies in #23 of OLAP query I appended in TPC-C++(SF=4) under a decoupled storage HTAP system; the OLAP query was only executed in the OLAP-side subsystem.

Measurement items	OLAP clients	OLTP clients 16	OLTP clients 24
Cycle occurrence rate (%)	8	41.8	49.1
	16	41.9	51.1
	24	34.1	51.9
The number of unique OLAP queries contained in a cycle	8	66	84
	16	93	135
	24	85	162
The number of finished OLAP queries	8	158	171
	16	222	264
	24	249	312
The number of cycles	8	84	219
	16	477	2304
	24	286	12319590

3.1.4 Transaction order for serializability

I revisit serializability conditions as the first goal in my proposed method before considering aborts, realistic implementations, or OLTP/OLAP performance. (If the examples regarding the relation between a multiversion conflict serializability and HTAP scheduler described in the previous section are sufficient, proceed to the next section.)

Adya et al. [ALO00] could determine whether a multiversion history of full schedule [Pap86] is serializable by analyzing the DSG derived from the transactions and conflicts (ww-/wr-/rw-conflicts). The serializability DSG verifies that the acyclic property is contained within the multiversion conflict serializability (MCSR) class of non-full schedule [WV01] . If a history H over $\{T_1, T_2, ..., T_k\}$ is acyclic in DSG(H), H could be equivalent to a serial monoversion history over $\{T_1, T_2, ..., T_k\}$, which indicates no interleaving of the operations mapping the same

versions of different transactions. Each direct dependency edge $T_a \rightarrow T_b$ in SDG(H) means that at least one of T_a's operations precedes and conflicts with one of T_b's. Therefore, in the serial monoversion history, all operations of T_a appear before any operation of T_b because any two conflicting operations in the serial monovertion history are arranged in the same order of conflicting operations in H. Suppose there is a cycle $T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_k \rightarrow T_1$ in DSG(H). The cycle implies that each of $T_1, T_2, ..., T_k$ appears before itself in the serial history H, so that no cycle can exist in DSG(H). That is, conflict serializability needs to guarantee a conflicts-based partial order obtained from DSG(H). Although considering the commit-operation order or the concurrency of transaction lifetimes as not being primary conditions for serializability, read/write/commit protocols can restrict the range of any transactions involving conflicts and serializability certifications, which will improve system performance.

Now consider the history described in Section 3.1.3 such as read-only anomaly example in an HTAP schedule. History h is not serializable because a transaction order could not be obtained from the cycle $T_3 \rightarrow T_2 \rightarrow T_1 \rightarrow T_3$ in DSG(h) by read-only transaction participation. However, if the read protocol of T_3 chooses a previous version of Y, where the write version order $Y_0 \ll Y_1$ is given by the order of history order $W1(Y0) \prec_H W1(Y1)$ of SI-based OLTP-side protocols (SI-W) and the version order cannot be exchanged, then T_3 can read Y from T_0. Such a T_3 operation $R3(Y0)$ can be commuted to be equivalent to a serializable history, for which T_0 is the last transaction preceding T_3 that writes any version of Y. I describe such a history h as follows.

$$h : R_3(X_0)\ R_3(Y_0)\ R_2(X_0)\ R_2(Y_0)\ R_1(Y_0)\ W_1(Y_1)\ W_2(X_2)$$

DSG(h) has edges $T_3 \rightarrow T_1$, $T_2 \rightarrow T_1$ and $T_3 \rightarrow T_2$ derived from all the conflicting operations $R_3(Y_0) \prec_H W_1(Y_1)$, $R_2(Y_0) \prec_H W1(Y_1)$ and $R_3(X_0) \prec_H W_2(X_2)$, respectively. DSG($h$) has a topological sort T_3, T_2, T_1 and h' is equivalent to a serial history executed in the order of $T_3\ T_2\ T_1$.

In addition, even if T_3's operations $R3(X)$ and $R3(Y)$ read the *successor* versions of the version order $X_0 \ll X_2$ and $Y0 \ll Y_1$, a serializable history h' can be also obtained as follows.

$$h' : R_2(X_0)\ R_2(Y_0)\ R_1(Y_0)\ W_1(Y_1)\ W_2(X_2)\ R_3(X_2)\ R_3(Y_1)$$

The conflicting operations $R_2(Y_0) \prec_H W_1(Y_1)$, $W_1(Y_1) \prec_H R3(Y_1)$ and $W_2(X_2) \prec_H R3(X_2)$ show edges $T_2 \rightarrow T_1$, $T_1 \rightarrow T_3$, $T_2 \rightarrow T_3$ in DSG(h'), respectively. A serial his-

tory executing operations in the order of T_2, T_1 and T_3, derived from a topological sort of DSG(h'), has equivalent conflicting operations in h'.

An OLAP read protocol that selects appropriate versions is required for HTAP systems aiming to minimize the impact of OLAP participation. Histories h and h' could achieve serializability without aborting transactions via the participating read-only transaction T_3. However, T_3's read operations do not know how best to choose versions. To be able define readable versions, OLAP read-only transactions have to be given conflict information (wr-/ww-/rw-dependency edges on DSG) by OLTP-side components. A version order can then be derived from ww-conflicts under SI-W in OLTP-side write protocol. OLAP-side serializability simply achieves finding appropriate versions that lead to a serializable history from the OLTP conflict information. I therefore focus on arranging the transaction model selecting versions for read-only transactions appropriately: Thus, OLTP can execute write transactions without synchronizing OLAP read-only transactions.

For the case that of OLAP always reading the most recent committed data (it is equivalent to SI), HTAP systems cannot achieve serializability without devising a theoretical framework around this activity. I focus on a serializable view constructed for the OLAP-side components without aborts, which selects appropriate versions to achieve conflict serializability, by using conflict information sent from OLTP-side components. This means that the methods need not send the read set of OLAP read-only transactions to the OLTP engines and does not need to synchronize the certifier's commit between OLTP and OLAP nodes. If the OLTP-side certifier (eg. 2PL) checked for overwrites (rw-conflicts) in the OLAP read set and the transactions were aborted or waiting to start, this would cause potential performance problems and implementation constraints caused by achieving HTAP system serializability.

Figure 3.2 shows an example of how my protocols work, with transactions T_1, T_2, T_3, T_4 and T_5 executed in OLTP-side components and a read-only transaction T_6 reading the installed data item using the information obtained through log-shipping, etc. The blue arrows from transactions T_1, T_2, T_3, T_4 and T_5 indicate (wr-/ww-/rw-) conflict edges on DSG within OLTP transactions. In this figure, for a given transaction T_a, B_a is written as $Begin(T_a)$ and E_a is written as $End(T_a)$. A line from B_a to E_a represents a theoretical transaction's lifetime (in fact, the begin operation have to be induced by the versions read by a read-only transaction, even though the physical time at which the read-only transaction began is current time in theory). The dashed line of T_5 indicates that the interval T_5 will include executed operations. The OLAP-side components have no means of knowing about the current running transaction T_5, which will write some data items after the OLTP logs are decoded at information-cognition

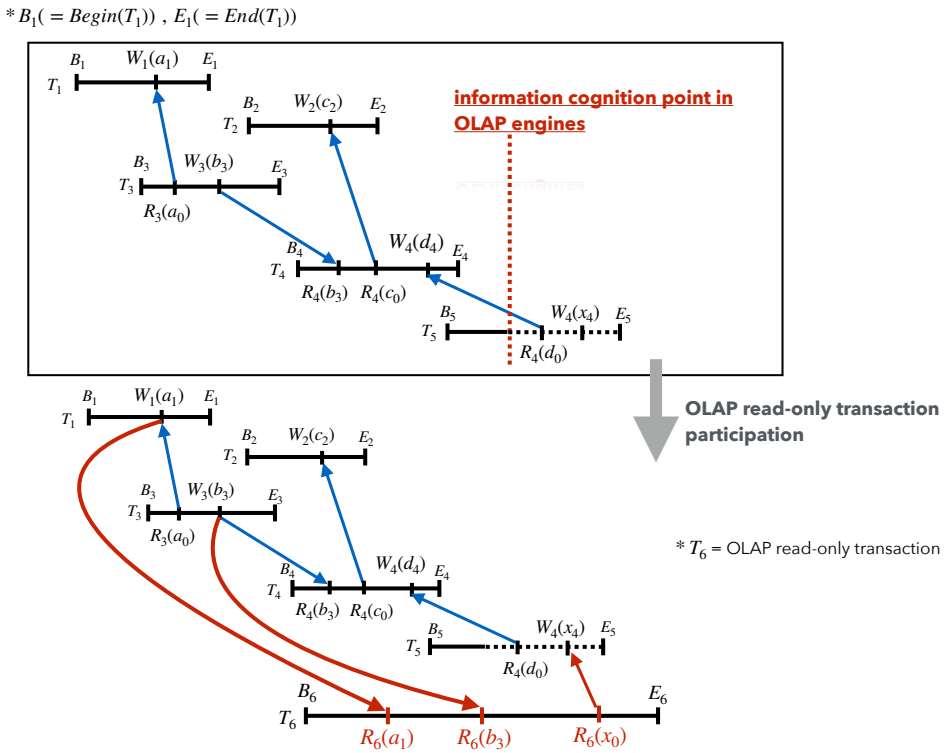

Figure 3.2: Example of a read-only transaction selecting a serializable view (serializable snapshot) in my model.

point in OLAP side. The OLAP read-only transaction T_6 is not aware of whether or not the successor version of data item T_6 is ready to be read. Even if T_6 waits (reader-wait) for the ending of T_5, the reader-wait cannot change the situation arising from overwrites (rw-conflicts) because new transactions will start continuously on the OLTP side. Moreover, even if T_5 waits *writer-wait* for the finish of T_6, the writer-wait will degrade OLTP performances and new read-only transactions will be starting in OLAP-side components. Therefore, HTAP systems cannot adopt reader-wait or writer-wait. Figure 3.2 shows the situation where T_5 writes X as the successor version of X_0 in T_6's read set, with the red arrow from T_6 to T_5 representing rw-conflicts from OLAP read-only transactions to current OLTP transactions. My model assumes there are always immediate rw-conflicts in currently running write transactions from OLAP read-only transactions. Therefore, T_6 targets write transactions that have no continuously connecting conflict edges from the current running write transaction. The red arrows from T_1 and T_3 to T_6 represent wr-conflicts mapping the target transactions and transactions which are not concurrent with currently running transactions. Although the lifetime of T_6 appears longer than the

actual execution time, this is no problem for serializability in MVCC theory, with $Begin(T_6)$ being the operation immediately preceding the first read operation.

If OLAP read-only transaction T_6 simply reads the most-recent committed version c_2 or d_4, read-only anomalies will occur via continuously conflicts from T_6 because DSG cycles $T_6 \rightarrow T_5 \rightarrow T_4 \rightarrow T_6$ or $T_6 \rightarrow T_5 \rightarrow T_4 \rightarrow T_2 \rightarrow T_6$ will arise. Read-only transactions wanting to read data items c or d can achieve serializability by reading the previous versions c_0 and d_0 from outside a transaction region composed of transactions T_5, T_4 and T_2 derived from $T_6 \rightarrow T_5 \rightarrow T_4 \rightarrow T_2$. Moreover, they can read most-recent versions a_1 and b_3 under a serializable transaction region conflict edges do not continue from concurrent OLTP transactions with information cognition points, which can be determined at a time, etc. of decoding logs, in OLAP nodes (subsystems). Conflict information $T_4 \rightarrow T_2$ can be sent from OLTP-side components because T_4 and T_2 are committed transactions. With respect to the information-cognition points, possible conflict information $T_6 \rightarrow T_5$ is regarded as always caused by the existence of initiated transactions on the OLTP side. Because systems do not know which data item transactions T_6, T_5 will read or write; I do not assume a deterministic execution framework requiring read/write information in advance. Using only the timestamp and conflict information in OLTP-side components, read-only transactions intruding into the history can enable a serializable history without theoretically unavoidable aborts and waits. Noting that OLTP transactions are executed under serializability in VOCSR. In addition, next section shows a model optimizing RSS by appending SSI properties on the OLTP side.

3.2 Problem statement

In Section 3.2, I illustrate develop a problem statement for serializability in decoupled storage HTAP systems. In the case of a naïvely configured architecture for decoupled storage architecture, there are problems related to consensus (synchronization) and communication costs, (among others) in a distributed environment aiming to assure serializability. The naïve method assures serializable isolation by using bidirectional synchronous communication (such as 2PL). The OLTP-side subsystem initially sends a WAL and the OLAP-side subsystem sends back the read-only transaction-related information it has read from data items in that log. Next, the OLTP-side subsystem verifies the returned information against its own write transactions, and returns information including read permissions and commits to the OLAP-side subsystem. My proposed method also guarantees serializable isolation by sending an RSS from the OLTP-side subsystem to the OLAP-side subsystem. That is, the OLTP-side subsystem supplies WALs and

RSSes and the OLAP-side subsystem processes only the read-only transactions in the WALs and RSSes. My decoupled storage HTAP system sends write-ahead log (WAL) and snapshot information via unidirectional communication, aiming to achieve reduced communication costs for ensuring serializability, as shown in Figure 3.3.

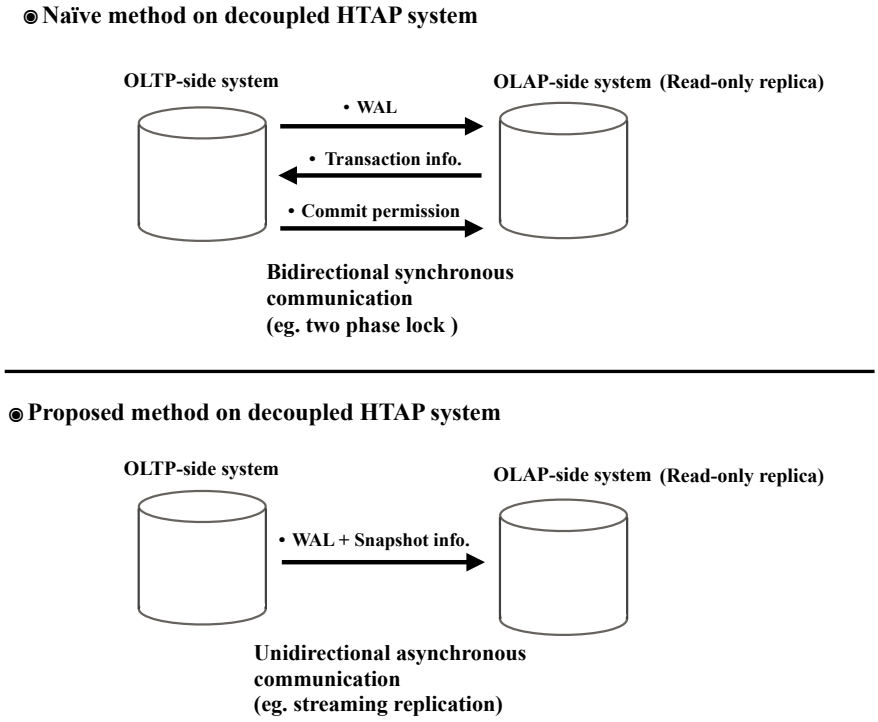

Figure 3.3: Conceptual diagram comparing a naïve method and the proposed method with respect to assuring serializability in decoupled storage HTAP systems.

The upper part of Figure 3.3 shows the bidirectional synchronous communication of a decoupled storage, which involves tasks HTAP system such as creating read-only replicas. The read-only replica contains the relevant information about read-only transactions used in protocols based on distributed transactions. For example, 2PC checks whether an abort or a commit has occurred in OLTP subsystem before completing a read-only replica for communication as the read set of the read-only transactions. If it finds single rw depenencies, either the writer of or reader in the OLTP subsystem will need to be aborted for serializability reasons, using a mechanism such as 2PL in a single-node system. SSI-based protocols can accept a wider transaction history than the protocols of 2PC because SSI theory incolves two successive rw dependencies

whereas 2PC responds to individual rw dependencies. However, the serializability cost can be substantial for HTAP workloads, caused by OLAP having to read a large proportion of the data set. Large read-sets can easily arise with rw depenencies derived from overwriting only small amounts of data, which leads to many aborts and waits in transactions. For this reason, very few decoupled storage HTAP systems must adopt the bidirectional-synchronous-communication approach if it aims to guarantee serializability. In decoupled HTAP systems, the OLTP-side subsystem can use conventional serializable methods of MVCC for executing the transactional part of HTAP workloads, but this does not guarantee serializability of the analytical part in terms of performance deterioration caused by aborts or waits, as described in Section 2. Even though the OLTP side could achieve serializability, the OLAP side can generates anomalies under the CC offered by MVCC unless there is some method of globally managing the entire transactions in the HTAP system. Decoupled HTAP systems would like to avoid sending back large read sets from OLAP to OLTP, so that the communication costs are incurred only in one direction: namely, from the write-optimized system to the analytical system, such as with asynchronous log-shipping replication sending WAL information. For unified storage systems, OLAP may execute transactions under SI by pre-labeling them as read-only when optimizing to improve performance. A methodology for guaranteeing serializable isolation of read-only replicas while limiting performance deterioration is the aim.

The lower part of Figure 3.3 depicts my proposed approach for decoupled storage HTAP systems. I presuppose the availability of histories in the form of read-only replicas for use with an architecture of asynchronous and unidirectional communication. This research regards the OLAP-side history as the global history of participating read operations in the OLTP-side history, and the *HTAP history* implies a global history that represents the operations communicated via unidirectional communication (such as streaming replication). Serializability is achieved when the global history can obtain a partial ordering of transaction derived from the snapshot information that I call RSS. In Section 4.1, I show that my model's RSS assures serializability by reading older versions than simply reading a naïve snapshot. In addition, I can certify that the read-only transactions of my model do not affect the global history even though the read operations have joined the original OLTP history (Theorem 1) or have been removed from the global history (Corollary 1).

A decoupled storage HTAP system comprises a system where the OLTP and OLAP subsystems are isolated from each other but the histories of OLTP-based and OLAP-based operations are maintained. A global history is produced by integrating the OLAP and OLAP histories, which contains a partial ordering of operations. Communication costs between the OLTP and

OLAP subsystem are increased by aborts or waits in transactions when managing the serializable transaction order. In particular, collecting information confirmed by bidirectional synchronous communication in a distributed system would worsen the system performance even more for a decoupled architecture than for a unified architecture, and HTAP systems users would be reluctant to have to wait during operations involving real-time analysis. For this reason, decoupled storage HTAP systems aim to avoid bidirectional synchronous communication (such as 2PL), but otherwise, the absence of such a global transaction manager may induce anomalies if serializability is not guaranteed. An OLAP read-only transaction under SI accesses data items (called snapshots) that reflect the most recently committed versions of data items written by OLTP transactions committed in the OLTP-side subsystem. I also define anomalies when naïvely sending and reading snapshots in Section 3.1.3.

Chapter 4

Serializability of HTAP systems

4.1 Read-safe snapshots (RSS)

Ideally, a DBMS would suffer little performance degradation by aiming for serializable isolation. In Section 2, I described how that previous studies have suggested SI-based mechanisms for OLAP read-only transactions, regardless of whether they assure serializability. Traditionally, guaranteeing serializability in a replicated system was more desirable than tolerating read-only anomalies under SI, but the naïve methods adopted were far from optimal as utilizations for read-only replica, because of the performance problems arising from aborts or waits when taking and sending naïve snapshots, as shown in Section 3.1.3.

In this section, I introduce my proposed model for constructing a view, in the form of an RSS, that assures serializability whenever read-only transactions join. I assume the following in my proposed model. First, the OLAP-side subsystem should execute transactions being as read-only. Second, the read-only OLAP transactions should only read RSSs. In other words, a read-only replica in the OLAP-side subsystem can only run transactions as read-only, without any pre-labeling identify transactions as either OLTP or OLAP from the beginning, and the reading target of read operations in an incoming transaction will be limited to reading the RSS only. The read operations in OLAP transactions does not assume SI-V, implying that the reading target does not read the most-recent versions always, but the execution of OLAP transactions will be wait-free and abort-free. HTAP systems applying my model will obtain the following benefits.

- In my model, read-only transactions processed on the OLAP side do not need to perform validations of write transactions such as finding conflicts that might cause a writer or reader to abort an OLTP transaction, because the history assures serializability by reading

only the prepared view, within which read-only anomalies do not occur.

- Read-only transactions do not make OLTP transactions wait by reading only prepared views, nor do they have to wait to read the views created in background processes. My model is assumed to be nondeterministic and a system applying my model need not know what or when concurrent transactions will write/read and commit/abort in OLAP read-only transactions.

I assume that a nondeterministic scheduler processes and accepts/rejects the operations step by step from the beginning of a history. Generally, almost the DBMSs works by the nondeterministic scheduler. Therefore, I arranged that my models can be applied to such generic DBMSs.

Prerequisite summary is follow; (a) OLTP-side scheduler generates serializable history contained in class VOCSR (VOCSR presupposes that version order can be given by any protocol, for example given from OLTP-side history by using SI-W protocol), (b) the flow of log-shipping regarding serializability is only from OLTP side to OLAP side, which sends operation information, etc. of each OLTP transaction from beginning to the end ("end operation $End(T)$" contains it's commit and finished writing transaction information to storages, and "begin operation $Begin(T)$" is induced by a first read/write operation in a transaction.) , (c) OLAP side recognizes OLTP-side operation information by decoding the shipped logs and the order of log regards as preserved by system implementation. My model assumes that OLAP read-only transactions read RSS: OLAP transactions are read-only, read-only transactions are able to read RSS, therefore OLAP transactions read RSS and achieve serializability. To consider delay, etc. of log-shipping, I create the model of RSS to be able to determine by any prefix of a history that OLAP-side subsystem (or called the OLAP-side components of an unified storage HTAP system) perceives OLTP-side history as follow; (1) ongoing OLTP-side history H_{oltp} contains the commits of two transactions:

$H_{oltp} : R_2^{(t)}(X_0)\ R_2^{(t)}(Y_0)\ R_1^{(t)}(Y_0)\ W_1^{(t)}(Y_1)\ C_1^{(t)}\ W_2^{(t)}(X_2)\ C_2^{(t)}$,

(2) perceived (log-decoded) OLAP-side history H_{olap} recognizes the information perceived point (log-decoded point) as a current prefix up to $W_2^{(t)}(X_2)$ in H_{oltp}, and therefore H_{olap} contains only one commit operation, which implies the delay derived from physical transmission between OLTP-side and OLAP-side components:

$H_{olap} : R_2^{(t)}(X_0)\ R_2^{(t)}(Y_0)\ R_1^{(t)}(Y_0)\ W_1^{(t)}(Y_1)\ C_1^{(t)}$,

(3) any read-only transaction reads a view of data objects (or called snapshot) by selecting a version called RSS. The explanations of RSS show in next section and after. Optimization

summary (the detail is explained in Section 4.1.2) is follow; (i) OLTP side extracts the dependency edges of conflict operations and sends the dependency edge information to OLAP side. This is because OLTP side needs to extract the dependency information for the serializability computation of DSG on VOCSR, and therefore it is more efficient to send the OLTP-side information to the OLAP side and use the information instead of re-extracting them from the conflict operations on the OLAP side. (ii) Based on that, to read the most recent versions as much as possible, send Begin/End information from OLTP side to OLAP side. This is because a back-forward edge can be caused from a concurrent rw anti-dependency by SI to recognize information by the OLAP side that such anti-dependency edge may come from running OLTP transaction. (iii) When OLTP side adopts SSI, RSS construction for read-only transactions can be simplified by the property of not creating dangerous structure. The simplification is proposed as an algorithm whose theory is verified in Section 4.1.2.

In fact, as long as systems can make the theory accomplished, anything is good in terms of implementations. I think the two arguments of a theory and an implementation should separate out. In order to cover wider space of serializable history on theory, the section of theoretical framework is formalized as the serializability space of VOCSR which realistic systems can't have implemented. Moreover, the section of algorithm presents the theoretical optimization of the framework by using SSI of the most wide serializable space implemented in realistic database systems.

The distinction between the two of OLTP and OLAP is for convenience of implementation. OLAP can be conceptualized as a part of read-only transactions in the transaction theory of concurrency control. RSS is available for many systems regardless of HTAP. However, HTAP systems would be able to achieve serializability while maintaining higher performance than a typical database system of OLTP. Therefore, I have formalized my model for HTAP in previous sections.

In the case of decoupled storage HTAP systems, my model incurs costs in sending in unidirectional information from the OLTP to OLAP and add by the OLAP side having to construct views. It should be noted that the view construction time depends on the implementation and the workloads. If view construction slows, the OLAP data freshness worsens. To address this issue, I suggest the methodology of RSS construction for reducing the view construction cost by using SSI in Section 4.1.2.

4.1.1 Theoretical framework

Here I explain how the serializability of HTAP histories is achieved. VOCSR assures serializability on the premise of a history of full schedule derived from the partial-order relation in multiversion. The partial order is determined by read/write operations. An HTAP history comprises the union of histories from the OLTP-side subsystem and the OLAP-side subsystem. The partial order of an HTAP history is also determined in terms of the participating read operations in the OLAP-side history within the OLTP history. The operation of OLAP reading something expresses the idea that a data-item version is written in the OLTP-side history and can thereby decide the order on OLTP-side history, because the OLAP-side history has "read-only" as a condition. Then, the two histories that summarize the operations of OLTP and OLAP can represent a single history (HTAP history) the derived from the partial ordering. The original OLTP history will have achieved serializability (in the serializability class of VOCSR), and I aim to show that the single history formed from the union of OLTP and OLAP histories also assures serializability in the VOCSR class. Moreover, I show that a history also achieves serializability if the read operations in an OLAP transaction are removed from the serializable history. In such circumstances, pre-labeling will no longer be necessary.

In this section, Figure 4.1 and Definition 2 are used to illustrate the read targets for the original OLTP-side history. I denote as any (an arbitary) read-only transaction that reads an RSS rather than an OLAP read-only transaction, hence I write a history without pre-labeling OLTP or OLAP. In Theorem 1, I prove that the whole history (comprising RSSs and the read operations that read RSSs) satisfies serializability in the VOCSR class. In Corollary 1, I show that a modified history, where the operations in a transaction that reads RSSs have been removed from the whole history, also achieves serializability. This theorem and corollary provide the foundation for being able to eliminate pre-labeling in OLTP or OLAP.

To obtain a set of transactions contained in VOCSR, the RSS is defined as the set of OLTP transactions having unreachable dependency edges in the DSG, given a prefix of the OLTP history, as shown in Figure 4.1. The sets of unreachable transactions can be determined because a new dependency edge derived from direct conflicts cannot occur between committed transactions in a history. In a decoupled storage HTAP system, an OLAP read-only transaction can recognize the operations within a history prefix at the time the decoded transaction information of logs (such as WAL) are sent from the OLTP side. The figure shows that dependency edges can be either reachable or unreachable between committed *OLTP transactions* in the history prefix.

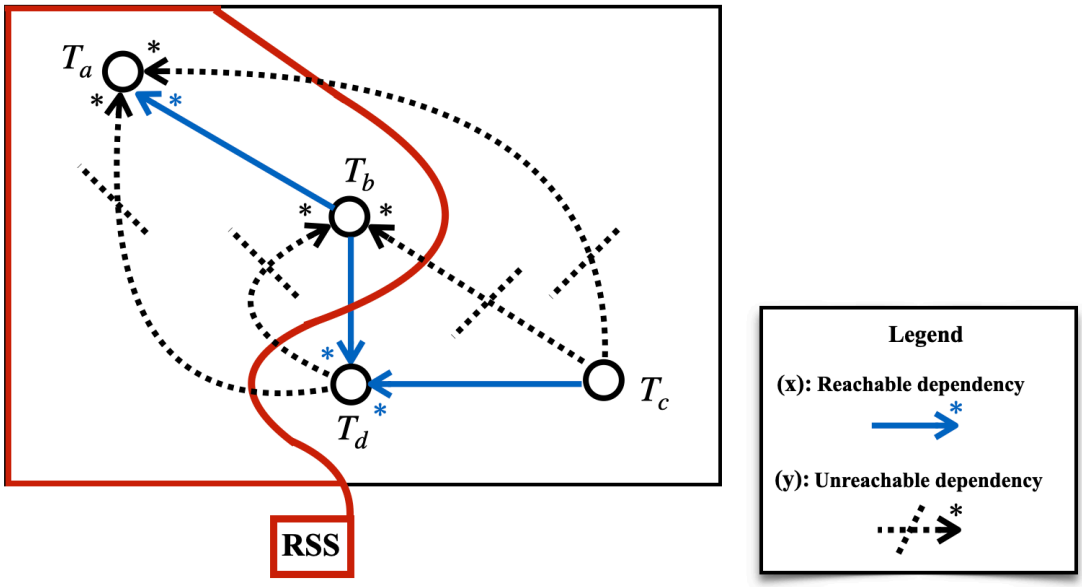

Figure 4.1: Conceptual diagram of an RSS and the dependency graph DSG associated with a given history from the OLTP-side subsystem. The round marks represent the nodes of *OLTP transactions* satisfying VOCSR and the arrows represent the dependency edges between the committed transactions. (x) represents a reachable dependency edge, and (y) represents unreachable dependency edges, where the dotted-line arrow indicates that transactions that are unreachable from transactions in the direction of the arrow do not occur in the history. From any read-only-transaction's point of view, T_a and T_b in the RSS can be considered as a transaction region protected against cycles in dependency-edge paths through T_c or T_d.

Definition 2 (Read-safe snapshots, RSS) *Let $\mathbb{T}$ be a set of committed transactions, $\mathbb{P}$ be a set of transactions contained in a history $\subseteq$ VOCSR, and $\mathbb{P} \subseteq \mathbb{T}$. I say that $\mathbb{P}$ is an RSS, if $\mathbb{P}$ satisfies the following condition:*

- *for arbitrary transactions $T_p \in \mathbb{P}$ and $T_q \notin \mathbb{P}$, T_p is unreachable from T_q: $T_q \not\rightarrow^* T_p$.*

In addition, I define a property with respect to the set of read-only transactions outside the RSS, whereby the read operations target the most successor versions in an RSS. I claim, in Theorem 1, that a multiversion scheduler achieving serializability can correctly accept operations from read-only transactions.

Definition 3 (Protected read-only transactions with respect to $\mathbb{P}$, $R(\mathbb{P})$) *Let $\mathbb{T}$ be a set of committed transactions, $\mathbb{P}$ be an RSS in $\mathbb{T}$, and $T \in \mathbb{T}$, where T*

- *is not contained in* $\mathbb{P}$,
- *has no write operations, and*
- *has only read operations that read the versions created by write operations in the most recent committed transactions in* $\mathbb{P}$.

I then say that T *is a protected read-only transaction (PRoT) regarding* $\mathbb{P}$ *and denote* $R(\mathbb{P})$ *as representing this property.*

I now introduce Lemma 1 for use in the proof of Theorem 1. The lemma claims that transactions meeting $R(\mathbb{P})$ cannot have an incoming wr-dependency edge (wr conflict) with respect to transactions excluded from the same set $\mathbb{P}$. Following this, Theorem 1 claims that a scheduler regarded as a hybrid of VOCSR and another scheduler can correctly accept a history as a serializable history.

Lemma 1 *For any* T_r *meeting* $R(\mathbb{P})$ *and* $T_q \notin \mathbb{P}$, T_r *is unreachable from* T_q*:* $T_q \not\to^* T_r$.

Proof 1 *I suppose that* T_r *is reachable from* T_q*. That is, there is a chain* $T_q \to^* T'_q \to T_r$*. Here,* $T'_q \notin \mathbb{P}$ *because otherwise,* $T'_q \in \mathbb{P}$ *would be reachable from* $T_q \notin \mathbb{P}$*, contradicting the fact that* $\mathbb{P}$ *is an RSS. A conflict with destination* T_r *only exists as a wr conflict, because no* T_r *includes write operations, from Definition 3. However, transactions having the a wr conflict with destination* T_r *can exist only for transactions contained in* $\mathbb{P}$*. Therefore,* $T_q \not\to^* T_r$ *holds.*

Theorem 1 *For a history* H *that applies committed projection, let* H *contain a* $\mathbb{P}$ *that composes RSSs. If a VOCSR scheduler* s *accepts a history* H' *for which the operations of* T_r *meeting* $R(\mathbb{P})$ *have been removed from* H*, then* H *is also accepted.*

Proof 2 *In* H'*, following Definition 3 and Lemma 1, the conflicts lost from* H *will be either*

(A) for a transaction $T_p \in \mathbb{P}$, $T_p \to T_r$ *or*

(B) for a transaction $T_q \notin \mathbb{P}$, $T_r \to T_q$,

where I suppose that dependency graph DSG for the given H *contains a cycle. The cycle in DSG(*H*) contains a dependency edge of either type (A) or type (B), because a dependency graph DSG from* H' *does not contain cycles, by Definition 2. For case (A), then* $T_p \in \mathbb{P}$ *and the cycle is* $T_p \to T_r \to^* T_p$*. Similarly, for (B),* $T_q \notin \mathbb{P}$*, and the cycle* $T_r \to T_q \to^* T_r$ *exists. In both cases, for* $T'_p \in \mathbb{P}$ *and* $T'_q \notin \mathbb{P}$, $T'_q \to T'_p$ *or* $T'_q \to T_r$ *would need to appear in the cycle, but that cannot hold, by Definition 2 and Lemma 1. That is,* H *cannot create a cycle in DSG(*H*). Therefore, if* H' *is accepted by* s*, then* H *is also acceptable.*

Corollary 1 *The opposite is also true: if H is accepted by s, then H' is also acceptable.*

Proof 3 *Because H' is obtained by removing only some of the conflicts from H, if a dependency graph DSG computed from a given H contains no cycle, a dependency graph DSG from H' will also be acyclic. Therefore, H' is accepted by s for cases where H is accepted by s.*

My model shows that a history will not affect scheduler acceptability if the scheduler has removed transactions that meet $R(\mathbb{P})$. Because this manipulation is recursively applicable, it does not matter how many times such $R(\mathbb{P})$ transactions are eliminated from the history. That is, provided the OLAP-side subsystem (or a read-only replica) always reads any of the RSSs constructed at the backend of transactional processing, scheduler acceptability for the primary replica is not affected. Finally, if the OLTP-side subsystem (or a primary) considers scheduler acceptability under traditional MVCC, achieving global serializability on HTAP (serializable HTAP) can be fully secured. In fact, the RSS would not be capable of including the most recent committed versions, because of OLTP retaining the starts of new transactions and reachable dependencies on DSG would occur. However, read-only transactions reading RSSs could always achieve serializability even if the read-only transactions joined at all times while processing OLTP transactions. Therefore, the execution of transactions that meet $R(\mathbb{P})$ will be abort-free and wait-free but this means that the most-recent versions written by committed OLTP transactions are not always the ones read such as SI.

As mentioned at the beginning of this Section 2, and also involving the HTAP history, my model shows that the read operations of an arbitrary read-only transaction (OLAP) meeting the property $R(\mathbb{P})$ are ordered in terms of the versions written by OLTP transactions, which implies that the HTAP history achieves serializability. The HTAP history of OLAP reading RSSs can trivially eliminate the read operations of OLAP read-only transactions. Moreover, this manipulation is recursively applicable any number of times. Therefore, pre-labeling is no longer necessary and the RSSs can continue to be built dynamically. In Section 4.1.2 next, I introduce an algorithm for constructing RSSs. Considering that the overhead of tracking dependency edges can be large when VOCSR is assumed in the OLTP-side subsystem, I change the assumption being made to the SSI alternative, and then verify whether my algorithm is theoretically correct.

4.1.2 An RSS construction method on SSI

In this subsection, I propose and investigate a construction method for RSSs. The model discussed in Section 4.1.1 showed that read-only transactions could read versions that ensured se-

rializability. However, a straightforward implementation would require information from each conflict and tracking a dependency graph. Therefore, a naïve implementation of my model could cause potential performance problems because the implementation would require multiple complex steps. For example, extracting conflicts included in the nonconcurrent state for the OLTP-side subsystem increases the tracking paths in dependency graph DSG and would require garbage collection of direct conflict information to clean up stale RSSs. I should consider how to implement my model without the OLAP participation affecting the OLTP performance significantly.

As described in Section 2, many previous studies have been based on SI. I therefore focus on using SSI as a representative example, because my model postulates serializability on the OLTP side. For the implementation, I propose a theoretical framework as an algorithm that optimizes RSS construction for SSI in OLTP. The algorithm has the characteristic of extracting only concurrent rw antidependency. Here, I assume the concepts in SSI (dangerous structure [CRF09, Cah27, FLO$^+$05] , SI-V and SI-W [SW00]) to develop a tractable algorithm. I present my algorithm in terms of an initial definition, where I define two types of transaction sets, and then describe the algorithm itself.

Definition 4 (Transaction state; $Done(p)$, $Clear(p)$) *Let H be a history under SSI, p be the prefix of H, and $\mathbb{T}$ be a set of transactions contained in H. I denote*

- $Done(p) = \{T \in \mathbb{T} \mid End(T) \in p\}$*; and*
- $Clear(p) = \{T_a \in \mathbb{T} \mid \forall T_b \notin Done(p)$ *such that* $End(T_a) \prec_H Begin(T_b)$ *}.*

The following Algorithm 1 utilizes my theoretical framework to construct an RSS via SSI.

Algorithm 1 Formal RSS construction algorithm under SSI.

(1) Contain entire $Clear(p)$ in RSS.

(2) Pick up any $T_c \in Clear(p)$.

(3) Pick up $T_u \notin Clear(p)$ and add to RSS if $T_u \rightarrow T_c$ exists.

(4) Repeat step (3) for all T_u

(5) Repeat steps (2)–(4) for all T_c

In the following, I aim to illustrate and investigate correctness of the algorithm. I also list some properties for use in the later discussion. First, I have self-evident properties derived from

Definition 4; $Clear(p) \subseteq Done(p)$ holds, $\forall T_a \in Clear(p), T_b \notin Done(p)$, $End(T_a) \prec_H Begin(T_b)$ holds and $\forall T_a \in Clear(p), T_b \notin Clear(p)$, $End(T_a) \prec_H End(T_b)$ holds. Next, I reiterate the definitions of dependency and dangerous structures from Fekete et al. [FLO$^+$05]. For $T_a, T_b \in \mathbb{T}$, if operations involving T_a, T_b (i.e., $Begin(T_a) \prec_H Begin(T_b) \prec_H End(T_a)$, $Begin(T_a) \prec_H End(T_b) \prec_H End(T_a)$, $Begin(T_b) \prec_H Begin(T_a) \prec_H End(T_b)$, and $Begin(T_b) \prec_H End(T_a) \prec_H End(T_b)$) are concurrent, and T_a, T_b satisfies $T_a \rightarrow T_b$, then I say that the direct conflict is a vulnerable dependency. Note that the conflict identified as a vulnerable dependency under SSI only exists as a concurrent rw dependency. For $T_a, T_b, T_c \in \mathbb{T}$, if $T_a \rightarrow T_b \rightarrow T_c$ holds and both dependencies are vulnerable dependencies, then this structure is a dangerous structure. An SSI scheduler would not accept a schedule containing this structure.

I now present three lemmas for verifying the appropriateness of the algorithm, where a history H is a committed projection and is acceptable to an SSI scheduler.

Lemma 2 *(SSI-1)*

For a history H and $T_a, T_b \in \mathbb{T}$ included in H, if $End(T_a) \prec_H Begin(T_b)$ holds, then $T_b \not\rightarrow T_a$ holds.

Proof 4 *Considering each conflict, for the case of an wr conflict that holds because of SI-V: the SI read protocol SI-V can only map each read operation to the most-recent committed write operations when the self-transaction begins. Therefore, $T_b \rightarrow T_a$ cannot arise from wr conflicts by $End(T_a) \prec_H Begin(T_b)$. For the case of a ww conflict, the lemma holds because of SI-W: SI-W was defined so that the write sets of two concurrent transactions are disjoint. SSI uses SI-W and obtains write version orders by the commit operation order in practice. Moreover, $T_b \rightarrow T_a$ cannot hold for ww conflicts. For the case of an rw conflict, the lemma holds because of SI-V: if $T_b \rightarrow T_a$ had held, then $End(T_a) \prec_H Begin(T_b)$ would be a contradiction by attempting to read the most-recent committed versions. Therefore, for each type of conflict, if $End(T_a) \prec_H Begin(T_b)$ holds, then the dependency is unreachable from T_b to T_a.*

Lemma 2 claims that direct dependencies do not arise from currently running (active) transactions to nonconcurrent completed transactions. In Lemma 3, I consider conflicts for the case that the begin operation induced from processing any operations precedes the end operation. If this case involves nonconcurrency, any direct conflict cannot occur against the commit operation order via Lemma 2. Otherwise, an rw conflict only exists in the reverse direction of transaction ordering between concurrent transactions. I can now define Lemma 4, which claims

that the property of Lemma 2 is related to $Clear(p)$ in Definition 4. The set of transactions in $Clear(p)$ is decided by the given prefix p of the history, where an arbitrary history prefix can be used.

Lemma 3 *(SSI-2)*

For a history H and $T_a, T_b \in \mathbb{T}$ included in H, let $Begin(T_a) \prec_H End(T_b)$. If $T_b \rightarrow T_a$ holds, then the dependency is a vulnerable dependency.

Proof 5 *If $Begin(T_a) \prec_H End(T_a) \prec_H Begin(T_b) \prec_H End(T_b)$, then $End(T_a) \prec_H Begin(T_b)$; here, $T_b \rightarrow T_a$ does not hold, according to Lemma 2. From the definition of vulnerable dependency, dependencies other than this case exist only as vulnerable dependencies. Therefore, the lemma holds.*

Lemma 2 shows that T_b must begin before T_a ends if a direct dependency from T_b to T_a exists. Lemma 3 describes its special case that T_b and T_a are concurrent, where only vulnerable dependency is allowed.

Lemma 4 *(SSI-3)*

For a history H and its prefix p, select an arbitrary $T_c \in Clear(p)$ and $T_u \notin Clear(p)$. If $T_u \rightarrow T_c$ holds, then it is a vulnerable dependency.

Proof 6 *Because $End(T_c) \prec_H End(T_u)$ according to $Clear(p)$ in Definition 4, $Begin(T_c) \prec_H End(T_u)$ holds. Therefore, it also holds for this lemma, via Lemma 3.*

I have now shown that the transactions contained in $Clear(p)$ are only reachable from transactions outside $Clear(p)$ through a vulnerable dependency, according to Lemma 4. Although a vulnerable dependency can exist in a dangerous structure (two successive vulnerable dependencies), an SSI scheduler cannot accept the second vulnerable dependency. If an SSI scheduler has accepted two successive dependencies with the first being a vulnerable dependency, then the second dependency cannot be a vulnerable dependency: ww or wr conflicts can be involved in the second dependency. The ww dependency and wr dependency can arise for nonconcurrent states without reversing the direction of the transaction order. That is, on such occasions, transactions that have created a second outgoing dependency edge are also contained in $Clear(p)$, according to the definition of $Clear(p)$.

Theorem 2 *(SSI-4)*

For an SSI history H and its prefix p, let $T_c \in Clear(p)$, $T_u \notin Clear(p)$, $T_v \in \mathbb{T}$. If $T_v \rightarrow T_u \rightarrow T_c$ holds, then $T_v \in Clear(p)$.

Proof 7 *$T_u \rightarrow T_c$ is a vulnerable dependency, according to Lemma 4, and a conflict therefore exists. By the contraposition of Theorem 2, $Begin(T_u) \prec_H End(T_c)$ holds. Moreover, $T_v \rightarrow T_u$ leads to $End(T_v) \prec_H Begin(T_u)$, because the SSI scheduler does not accept a dangerous structure (if $T_v \rightarrow T_u$ could exist, then an SSI scheduler must have accepted operations from nonconcurrent transactions). Therefore, $End(T_v) \prec_H End(T_c)$ holds and such a T_v can be contained in $Clear(p)$ only.*

I now prove the validity of my algorithm. Assume that $\mathbb{P} \subseteq \mathbb{T}$ is a subset detected by Algorithm 1. To be able to say that $\mathbb{P}$ is RSS, for any $T_p \in \mathbb{P}$ and $T_q \notin \mathbb{P}$, I need to show $T_q \not\rightarrow^* T_p$. To prove this by contradiction, assume $T_q \rightarrow^* T_p$. There must then be at least one direct dependency in the chain from a transaction outside $\mathbb{P}$ to a transaction in $\mathbb{P}$. By renaming the transaction if necessary, I can assume $T_q \rightarrow T_p$, where $T_q \notin \mathbb{P}$ and $T_p \in \mathbb{P}$. $T_q \notin Clear(p)$ obviously holds because of Step (1) in Algorithm 1. Consider two cases for T_p. First, in the case of $T_p \in Clear(p)$, $T_q \rightarrow T_p$ does not hold because of $T_q \in \mathbb{P}$ in Step (3) of Algorithm 1. Second, if $T_p \notin Clear(p)$, for $T_c \in Clear(p)$, then T_c such that $T_q \rightarrow T_p \rightarrow T_c$, must exist from Step (3) of Algorithm 1. However, $T_q \in Clear(p)$ must hold, according to Lemma 2, and therefore $T_p \notin Clear(p)$ implies a contradiction. Such a pair T_p, T_q cannot exist, which contradicts the assumption, because both T_p cases have been dismissed. Therefore, $\mathbb{P}$ is shown to be an RSS and Algorithm 1 is appropriate.

Figure 4.2 shows the relationship between the SSI properties and the set of defined transaction states $Done(p)$ and $Clear(p)$. Here, *undone* transactions (a complementary set to *done* transactions, $T \in Done(p)^c$) are not concurrent with *clear* transactions $Clear(p)$, by the read protocol SI-V. Conversely, transactions concurrent with *clear* transactions are possibly contained in $Done(p)$. I call such transactions in $Done(p)$ *Obscure transactions* ($T \in Done(p) \wedge \notin Clear(p)$). To reach *clear* transactions from *undone* transactions, two dependency edges via *obscure* transactions are essential. As shown in Figure 4.2, such essential features cannot arise as dangerous structures under SSI. *All transactions in the figure represent OLTP transactions*. T_j, T_m and T_q are uncommitted (*undone*) transactions. T_i and T_j show that the nonconcurrent relation between *clear* and *undone* transactions cannot give rise to rw dependencies. T_k, T_l and T_m show that dangerous structures cannot arise: if the SSI scheduler had accepted $T_k \leftarrow T_l$, the committed transaction T_k and T_l would be unreachable from any *undone* transactions via the *obscure* transaction T_l. Committed transactions contained in *clear* and *obscure* transactions cannot give rise to two successive rw dependency edges such as $T_p \rightarrow T_l \rightarrow T_k$ by SSI. This is because these OLTP transactions are executed under an SSI that does not involve the dangerous structures. Similarly, if $T_p \leftarrow T_q$ holds, the SSI scheduler

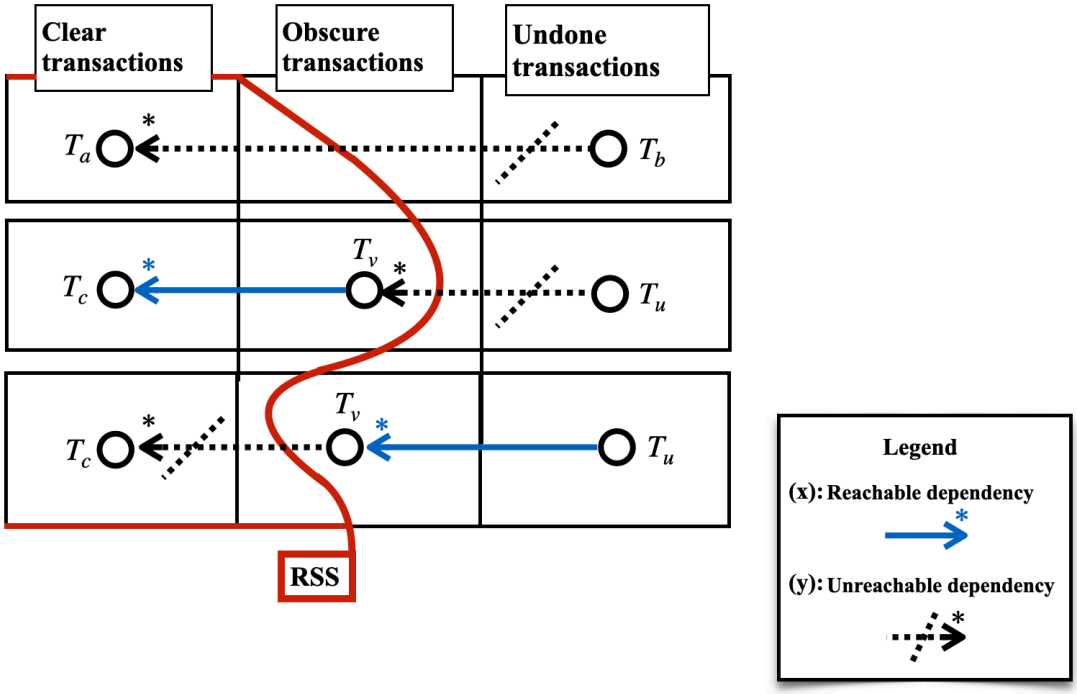

Figure 4.2: Conceptual diagram of RSS under a history based on SSI. (x) represents transactions that are reachable in the direction of the arrow as rw-dependency edges, and (y) represents transactions that are unreachable in the direction of the arrow as any dependency edges. T_a and T_b show that the non-concurrent relation between *clear* and *undone* transactions cannot give rise to rw-dependencies (shown in Lemma 2). T_c and T_v show committed transactions, and T_u shows uncommitted transactions. If an SSI scheduler had accepted $T_c \leftarrow T_v$, it would abort other transactions that would lead to $T_v \leftarrow T_u$. Similarly, if $T_v \leftarrow T_u$ holds, an SSI scheduler would not have allowed $T_c \leftarrow T_v$. SSI-based history can create RSS by looking at just one rw-dependency (not two or more consecutive ones) between two committed transactions of T_c and T_v.

would not have allowed $T_o \leftarrow T_p$. Therefore, the RSS would not contain T_p in spite of its being committed. This is because a cycle would arise if read-only transactions did read the write of transaction T_p. As a result, even if $T_p \leftarrow T_q$ did not arise, the RSS could have contained T_o because T_o would have been unreachable, such as $T_i \leftarrow T_j$. In summary, *undone* transactions cannot reach *clear* transactions. Moreover, *undone* transactions cannot reach *done* transactions having an rw dependency edge toward *clear* transactions. The *clear* and *done* transactions can then be contained in the RSS. Therefore, for any prefix in a history generated under SSI, the un-

reachable transactions from *undone* transactions can join the RSS, and read-only transactions can read the RSS to achieve serializability.

Algorithm 1 represents that any (an arbitrary) prefix in a history can be chosen dynamically on a HTAP scheduler. Therefore, I can assume that the prefix of a history is dynamically given from the OLAP-side subsystem via decoding the log information (such as WAL) sent from the OLTP-side subsystem when running in a decoupled storage HTAP system, and the prefix will be the current prefix, representing the history up to the time when the decoding function is completed. My model assume a single history without unifying every notations into SI, which is one history of the two when the OLTP side is SI and the OLAP side is not SI in a decoupled storage HTAP systems. The assumed behavior of the OLTP side depends on the starting position of $Begin(T_a)$, whereas the OLAP side is independent of the starting position of $Begin(T_a)$. I can now present a suitable dynamic algorithm for this process.

Algorithm 2 shows that SSI-based RSSs are dynamically constructed while both OLTP and OLAP subsystems are running in an example of a decoupled storage HTAP system. As events in the real world advance incrementally, the HTAP system can construct a series of RSSs dynamically. Algorithm 2 were written to enable viewing of the algorithm's dynamic and incremental operations as an invokable function (Line 2). Line 1 in Algorithm 2 shows that information about transactions outside this RSS (such as previous invocations of $Construct - SSIbasedRSS$) are regarded as uncertain in commit/abort state. Such information (written as $ExistTxns$) may include additional transaction information about operations and outgoing dependency edge of transactions from decoding the OLTP log in other processes (Lines 3–4). I also assumes that OLTP log decoding process is incremental and does not re-decode the log from the beginning. The decoding process saves the location as the prefix of the log until new information is sent to this algorithm, with the next process starting from the saved location. Lines 5-12 express that information from $ExistTxns$ splits into *Active* or *Done* transactions depending on whether or not commit/abort operations are included. A function extracts the information about each transaction from the decoded information (Line 5). If an abort operation is included, the algorithm musts eject the information for all transactions including the abort operation (Line 8). Line 12 shows that the information written as $PreviousOutsideRSSTxns$ is preserved for constructing the RSS in the next iteration, with other variables can sent for garbage collection.

Lines 13–18 express that partitions *Done* transactions into *Clear* or *Obscure* transactions. To realize Algorithm 1, *Begin* and *End* assume the timestamp of the first read/write and last commit/abort operations of the OLTP transactions (Line 13). Setting the first read/write oper-

Algorithm 2 Construct SSI-based RSS.

Input: Log : ← OLTP transaction information received from WAL decoding layer of OLAP-side components

Output: RSS

1: $PreviousOutsideRSSTxns \leftarrow \emptyset$
2: **procedure** CONSTRUCT-SSIBASEDRSS(Log)
3: $ExistTxns \leftarrow PreviousOutsideRSSTxns$
4: $ExistTxns \leftarrow Log$
5: **for** $existTxn \leftarrow$ Extract transaction information from $ExistTxns$ **do**
6: **if** $existTxn$ contains a commit operation **then**
7: $DoneTxns \leftarrow existTxn$
8: **else if** $existTxn$ contains an abort operation **then**
9: Eject transaction information of existTxn
10: **else**
11: $ActiveTxns \leftarrow existTxn$
12: $PreviousOutsideRSSTxns \leftarrow ActiveTxns$
13: $activeMinBeginOperation \leftarrow$ Extract the most precedent operation of $ActiveTxns.operations$
14: **for** $doneTxn \leftarrow$ Extract transaction information from $DoneTxns$ **do**
15: **if** $doneTxn.op.commit \prec_H activeMinBeginOperation$ **then**
16: $ClearTxns \leftarrow doneTxn$
17: **else**
18: $ObscureTxns \leftarrow doneTxn$
19: **for** $obscureTxn \leftarrow$ Extract transaction information from $ObscureTxns$ **do**
20: **if** $obscureTxn.dep$ has a reachable dependency edge toward a transaction contained in $ClearTxns$ **then**
21: $RSS \leftarrow obscureTxn$
22: **else**
23: $PreviousOutsideRSSTxns \leftarrow obscureTxn$
24: **for** $clearTxn \leftarrow$ Extract transaction information from $ClearTxns$ **do**
25: $RSS \leftarrow clearTxn$
26: return RSS

ation as $Begin(T)$ is required for dividing transactions in *Clear* or *Done*. OLAP can perceive the current running transactions in OLTP identified by $Begin(T)$ being sent from the OLTP-side subsystem to OLAP-side subsystem, thereby enabling the execution of OLAP transactions to be wait-free. In addition, setting the last commit/abort operation regarded as $End(T)$ is required for checking whether or not the current running transactions have completed (lines 15–18). These two operations (setting the first read/write as $Begin(T)$ and last commit/abort as $End(T)$) need to be linked timewise, which is not a problem because all timestamps are based on the clock in the OLTP-side subsystem. I intend that the $Begin(T)$ of an OLAP transaction should not depend on the physical start timestamps. To use the explanations of my model, the start position $Begin(T)$ of OLAP that appeared immediately before a first read operation $R(X)$ and reads the most recent committed version of a data item X written up to that point (the history prefix).

Lines 19–26 show that RSS construction is determined by whether or not a dependency edge from *Obscure* transactions to *Clear* transactions exists. After the process of Line 26, OLAP-side subsystem can remove old versions of data items contained in previous RSS construction procedures if the newest RSS constructed by the current procedure contains newer versions of the data items. In the next section, I describe the implementation of Algorithm 2 in an HTAP system which could offer SSI because the algorithm is based on SSI. SSI is a representative method for assuring serializability in a multiversion history under SI. In Section 4.3.1, I found that the system performances of both naïve SSI and my algorithm were an improvement on the SSI in a unified storage HTAP system.

4.2 The gap of implementation and serializability

In this section, I introduce a prototype implementation that expresses Algorithm 2 in a practical environment. I implemented the prototype on top of the PostgreSQL-provided SSI, because my algorithm assumes SSI for serializability with respect to OLTP. I implemented the algorithm for both unified and decoupled architectures [RCAA20] because my model is applicable to both. Although unified HTAP systems could have achieved OLAP serializability derived from traditional transaction theories, writer/reader aborts/waits (rw conflict blocking or safe-snapshot blocking) could have occurred with read-only transaction participation. As described in Section 2, in the case of serializable isolation for a decoupled storage HTAP system, a modern architecture must either send the read sets and write sets of concurrent transactions to each OLTP and OLAP subsystem while synchronously checking the read sets and write sets, or

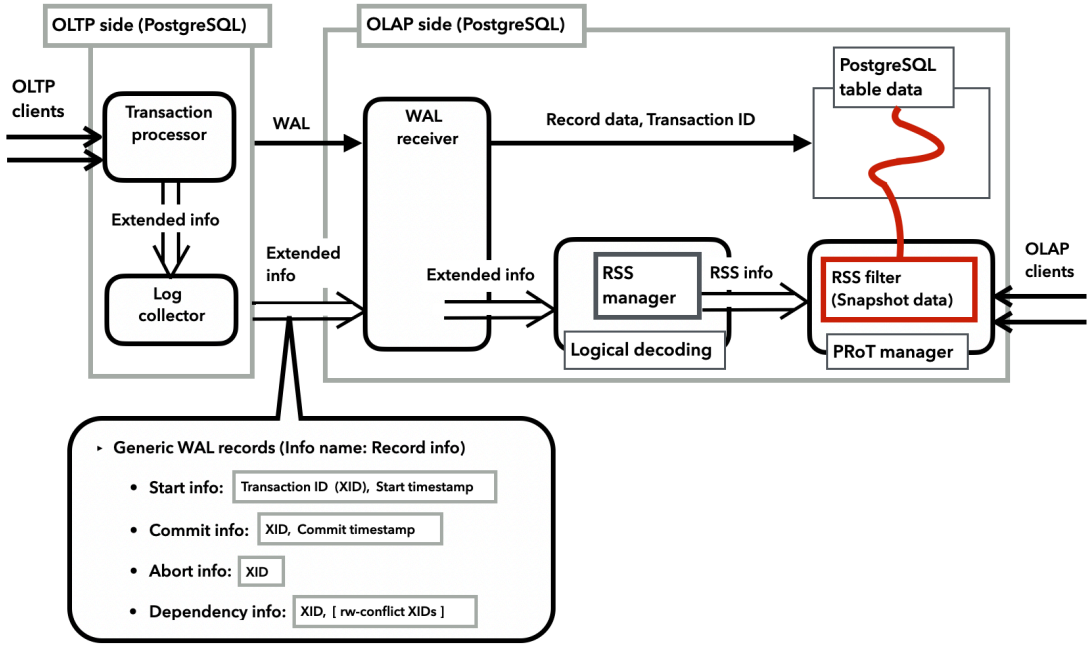

Figure 4.3: Decoupled storage (multinode) architecture overview. Twin arrows $\rightrightarrows$ represent clients. A single arrow $\rightarrow$ represents existing processing flows of PostgreSQL. An open arrow $\Rightarrow$ represents newly implemented processing flows.

would have to stop starting new transactions and let the OLAP replica wait for a consistent view. In fact, the idea of SafeSnapshots for a read-only replica was proposed in Streaming Replication [PG12], but has yet to be implemented. The PostgreSQL implementation uses SafeSnapshots only for unified storage architectures, but the method cannot always terminate within a fixed time interval and may have to wait an unexpectedly long time.

My model demonstrates that additional read-only transactions can combine with in the existing set of transactions to ensure serializability. Systems adopting the model are not forced to abort or wait for OLTP transactions and OLAP read-only transactions. I considered that my HTAP-compatible model could avoid such an inferior performance. Therefore, I implemented my decoupled storage HTAP system based on the asynchronous log-shipping replication technique that PostgreSQL offers and describe the implementation in Section 4.2.1 below.

4.2.1 Decoupled storage architecture

An OLAP node can construct an RSS by using only the timestamps in the OLTP node of a decoupled storage HTAP system. The OLAP side need not send back information (such as

a read set) for serializability in my model, but the OLTP side must ship rw antidependency and start/end operation orders to the OLAP side. Figure 4.3 shows my implementation based on PostgreSQL version 12.0 and the flow of information I added for serializability. In the following, I describe the components and custom-logic elements that play important roles in integrating my model into PostgreSQL's SSI. To realize the SSI theory, PostgreSQL detects rw conflicts between transactions for as long as any concurrent transactions are alive.

Dependency information. My implementation collects the detected outgoing rw dependencies and writes the dependencies as a WAL record with self-transaction IDs to construct an RSS immediately after any reader transactions with outgoing edges are committed. The log collector shown in Figure 4.3 extracts the outgoing rw dependencies that are relevant to my algorithm. The collected dependencies about a transaction are written in the form of the logical messages that PostgreSQL offers as generic WAL records. Each of the direct conflicts is expressed as an array of writer transaction IDs in the generic WAL record.

Start/end information. In addition, the commit, start and abort information about when each transaction starts (induced by the first operation) and ends is appended to the WAL. The written commit information is needed to construct the $Done(p)$ of transaction schedules from a regular WAL scan on the OLAP side. The start/commit information is required to create $Clear(p)$ on the prefix for active transactions that have started but not ended (committed or aborted) at the time when the WAL inspection is executed. For the management of active transactions, abort information is used to exclude other transactions from active transactions.

OLTP read-only transactions. Although the PostgreSQL implementation does not require a read-only transaction to have a transaction ID assigned, I assign one for read transactions because the DBMS does not know beforehand if the transactions are actually read-only. When merging the transaction histories of OLTP and OLAP, a cycle in the DSG could arise even though one of the rw conflicts exists only on the OLTP side. As a result, the OLTP-side implementation collects dependency information and start/end information about read-only transactions in addition to write transaction information.

Version preservation. The logical messages appended to WAL for constructing the RSS are sent by the asynchronous commit option for Streaming Replication. PostgreSQL is implemented to prevent multiversions from overwriting previous versions of data items. To maintain the previous version, updating a transaction creates a new tuple attached to a self-transaction ID and installs the tuple in storage. Stale tuples are deleted by two types of garbage collection, namely vacuum and heap-only-tuple (HOT). The vacuum method is explicitly invoked by user calls, but HOT is implicitly executed, irrespective of the user's intention. My implemen-

tation needs to protect the stale tuples from HOT actions by sending active-transaction-alive information from a read-only replica. This is called “hot-standby-feedback” in a PostgreSQL configuration.

I use the logical decoding offered by PostgreSQL for reading the information in generic WAL records. I reformatted the read replica in the Streaming Replication environment to enable this logical decoding to be used. The logical decoding is then executed via an external processing module at fixed intervals. The RSS construction invoker is an external module that retains snapshots and replaces them with RSSs as transactions. At regular intervals, the RSS construction invoker executes PostgreSQL’s logical decoding and various user-defined functions (UDFs) that I created to decode the generic WAL, to manage the dependency graph, and to construct the RSSs.

RSS manager. The decoded information is used to construct a brief transaction history in PostgreSQL shared memory. The history sequence comprises the start and commit times of transactions that have been started or committed by the time the invoker transactions are executed. The RSS manager constructs “Active”, “Done”, and “Clear” transactions by scanning the history using a hash table of transaction IDs. Active transactions conmprise transactions having start information only, whereas, in the previous history prefix, previous Active transactions are transformed to done or clear transactions if commit information existed at the time of scanning the WAL. If abort information has arrived at the time of the WAL inspection, transaction IDs are transferred from Active transactions to the set of transaction IDs awaiting garbage collection. The clear and done transactions are carefully managed in the RSS construction algorithm.

In addition, decoded dependency information is required for the construction of RSSs. I therefore implemented a dependency graph in the shared memory of PostgreSQL, which preserves the sent dependency information, as a vertex for the transaction ID and a path for the dependencies. If the dependency graph reveals paths from Clear transactions to Done transactions, the RSS contains the discovered transaction IDs. Prepared as a UDF, these RSS construction and dependency graph operations are invoked in a snapshot, replacing the transaction that the RSS construction invoker offers.

PRoT manager. To replace snapshots with RSSs, the snapshot-preserving transactions called from the RSS construction invoker must continue executing until the time that the next RSS construction has ended. The RSS and dependency-graph construction operations were implemented in the PRoT manager as UDFs. The PRoT manager invokes RSS and dependency-graph construction operations and receives the current RSS information as snapshot data for

PostgreSQL from the RSS manager. The RSS snapshot data are exported to transactions in the read-only replica as PostgreSQL's snapshot data.

4.2.2 Unified storage architecture

Although my algorithm can be efficiently applied to the decoupled storage architecture, other approaches to achieving serializability have not involved decoupled storage architecture, and existing HTAP systems ignore serializable isolation for better performance. In addition, the idea of safe snapshots on a read-only replica in [PG12] was not implemented in PostgreSQL. Because comparative methods such as SSI and SSI+Safe Snapshots have only been applied to unified storage architectures, I also implemented my algorithm as a unified storage architecture, to enable comparative evaluations of performance overheads from aborts and waits involving serializability.

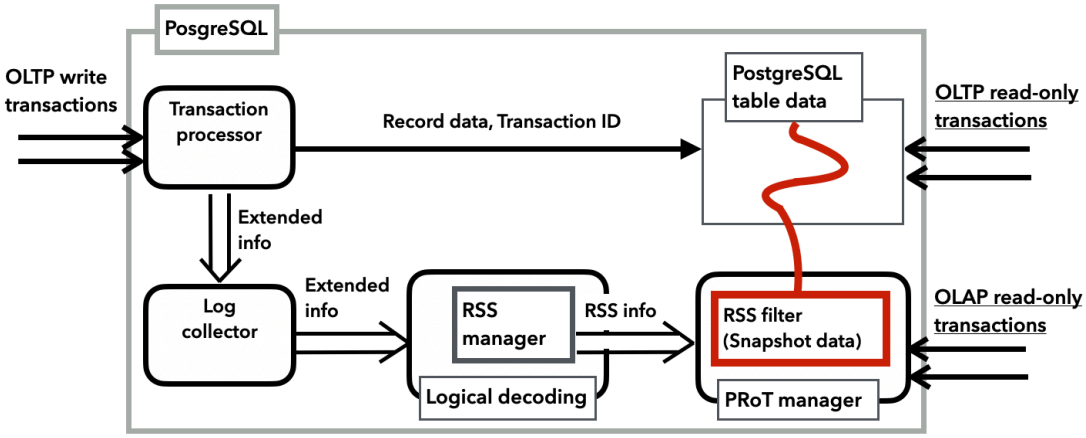

Figure 4.4: Single-node architecture overview

The unified storage system I implemented is illustrated in Figure 4.4. This system was developed from the multi-node architecture described in Section 4.2.1, with the main difference being the omission of the log-shipping replication. The versions that a RSS contains are robustly preserved by the snapshot-preserving transactions of the RSS construction invoker until the next RSS is constructed. This architecture needs to recognize when transactions are read-only for RSS to be used. Because analytical-use queries such as TPC-H benchmarks [Coub] are known to be read-only in advance, I modified such benchmark queries by appending a PostgreSQL command that confirmed the transaction as read-only. In addition, the implementation of unified storage architecture was modified to execute a transaction that could read RSS only

if the PostgreSQL read-only flag exists. Regarding OLTP read-only transactions, as presented in the TPC-C benchmark [Coua], although RSS is not used, read-only anomalies should not arise from the SSI as OLTP protocols, because I consider that the OLTP side would be used primarily for processing write transactions.

4.3 Evaluation

In this section, I report on my evaluation of the two prototype systems proposed in Section 4.2. My systems were evaluated in terms of the CH-benCHmark [CFG+11] of OLTPBench [1] [DPCCM13]. OLTP-Bench implements the CH-benCHmark derived from TPC-C and TPC-H benchmarks to evaluate DBMSs designed to serve both OLTP and OLAP workloads. All experiments were conducted with a scale factor of 100 (SF100; 100 warehouses in CH-benCHmark) for the benchmark test. OLTP transactions were always executed under SSI for serializability. Each test duration was 5 minutes and the warmup duration was 60 seconds after the initial data load of SF100 was completed. The total run time was about 25 minutes. Comparison systems were set up for each of the evaluation purposes associated with my two systems, as follows.

- First, I investigated the abort rate and performance compared with both of PostgreSQL's SSI and SafeSnapshots, noting that PostgreSQL can only support each serializable method in a unified storage architecture. I denote the application of safe snapshots (read-only deferrable transactions) in TPC-H queries as "SSI+SafeSnapshots" Executing OLTP transactions and OLAP queries under SSI is denoted as "SSI". My prototype system, where analytical queries derived from TPC-H were executed under RSS, is denoted as "SSI+RSS".

- Second, I tested the performance overhead of RSS construction against the SI that PostgreSQL offers in terms of a repeatable read-isolation (SI) level in a read-only replica. Each primary and read-only replica was launched by PostgreSQL instances on two servers, to represent a decoupled architecture. The primary and read-only replicas were operated on independently on separated machines. I denote the application of SI in the read-only replica as "SSI+SI". Finally, my system architecture, where the primary node uses SSI and the replica node applies RSS, is denoted as "SSI+RSS".

Environment. The experiments involving single-node architectures were run on a client node issuing transactions for the CH-benCHmark and a database node executing the transactions.

[1] `https://github.com/oltpbenchmark/oltpbench`

The client node was equipped with fmy Intel(R) Xeon(R) Platinum 8176 CPUs with 2.10 GHz processors having 28 physical cores (32-KB L1i + 32-KB L1d cache, 1024-KB L2 cache, and 39.424-MB L3 shared cache) and 224 logical cores, 512 GB of DRAM, and 440 GB of SSD. The storage performance of 4 KiB I/O was 35.0 MiB/s throughput, 9209 IOPS and 20 milliseconds (ms) latency for sequential writes, 2381 MiB/s throughput, 610000 (610k) IOPS, and 4 microseconds (μs) latency for sequential reads, 43.5 MiB/s throughput, 11.1k IOPS and 100μs latency for random writes, and 8791MiB/s throughput, 2251k IOPS, and 50μs latency for random reads. The database node had two Intel(R) Xeon(R) Platinum 8176 CPUs clocked at 2.10 GHz with 28 physical cores (32-KB L1i + 32-KB L1d cache, a 1024-KB L2 cache, and a 39.424-MB L3 shared cache) and 112 logical cores, 1 TB DRAM, and 440 GB SSD. All experiments were run on these nodes under the Ubuntu 18.04.3 OS and PostgreSQL 12.0. For the experiments involving a multinode system, the PostgreSQL primary and read-only replica were constructed using two of these database nodes, one for the OLTP-side system and one for the OLAP-side system. The OLTP-side storage performance of 4 KiB I/O was 67.3 MiB/s throughput, 17.2k IOPS, and 50 μs latency at sequential writes, 5941MiB/s throughput, 1521k IOPS, and 50 μs latency at sequential reads, 53.6 MiB/s throughput 13.7k IOPS, and 100 μs latency at random writes, and 2038 MiB/s throughput, 522k IOPS, and 4 μs latency at random reads. The OLAP-side storage performance of 4 KiB I/O was 63.5 MiB/s throughput, 16.3k IOPS, and 10 ms latency at sequential writes, 5208MiB/s throughput, 1333k IOPS, and 50 μs latency at sequential reads, 62.6 MiB/s throughput, 16.0k IOPS, and 10 ms latency at random writes, and 7156 MiB/s throughput, 1832k IOPS, and 20 μs latency at random reads. Schematic designs of the experimental setup are shown in Fig. 4.5. The database nodes in the figure are denoted as database node 1 for the OLTP side and database node 2 for the OLAP side.

As described previous section, my prototype implementation of unified storage architecture could select RSS if read-only flag exists in a query. The implementation of SSI+SafeSnapshots can also use SafeSnapshots when read-only flags exist in OLAP queries. In addition, the set-up of decoupled storage architecture was created by using log-shipping replication functions of PostgreSQL's streaming replication. OLTP and OLAP of CH-BenCHmark were issued from the client server (client node) to the database server as shown in Figure 4.5.

4.3.1 Serializability impact on OLTP and OLAP performances

I compared experimentally the serializability methods of SSI and SSI+SafeSnapshots under hybrid OLTP and OLAP workloads. The serializability of the SSI+RSS implementation described in Section 4.2.2 was confirmed by reading the previous version and using the example

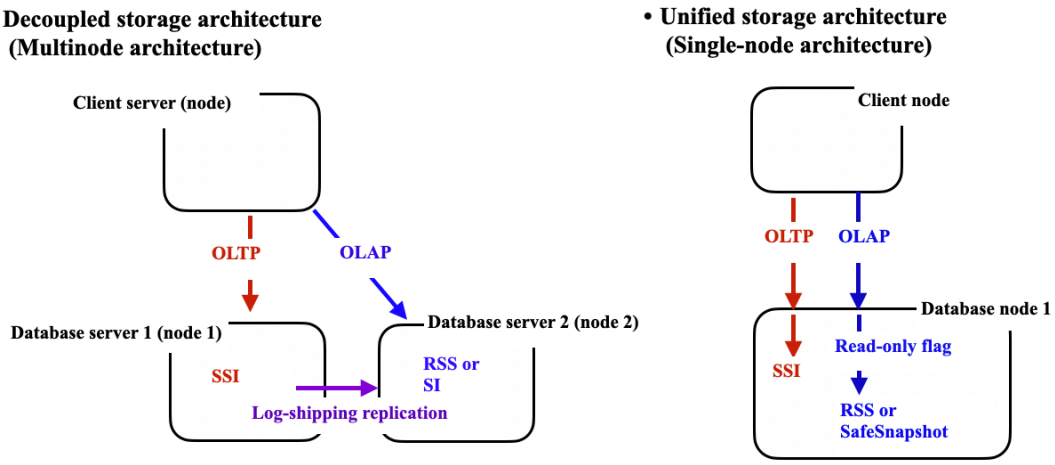

Figure 4.5: A schematic diagram of the experimental setup.

queries of Ports and Grittner [PG12] that illustrated a read-only anomaly in PostgreSQL. The reader-wait in SafeSnapshots (Deferrable Transaction [PG12]) was also confirmed by the example of a read-only transaction anomaly [FOO04]. In the example described in Section 3.1.3 for history H_{roa}, where the read-only transaction T_3 waits for the started concurrent transaction T_2 to finish, it must have waited until the time that T_2's commit command was issued. Such read-only deferrable transactions might affect concurrent write transactions in OLTP as much as in OLAP. Although these alternative methods might be affected by the validation cost of serializability, I considered that SSI+RSS might be able to avoid such costs derived from read-only transaction participation. Therefore, I investigated the individual performances of OLTP and OLAP, while varying the number of OLTP and OLAP clients.

The upper part of Figure 4.6 shows the change in the average (over 10 runs) OLTP performance when executing the CH-benCHmark as the numbers of OLTP and OLAP clients were increased from 1 to 48. The x-axis shows the number of OLTP clients and the y-axis shows the number of succeeded OLTP transactions per second. The OLTP throughput did not change much, even for 48 OLTP clients, and was affected more by the physical core limitations of my experimental environment. Similarly, the middle part of Figure 4.6 shows the average OLAP performance as the numbers of OLTP clients and OLAP clients were increased from 1 to 48. Here, the y-axis shows the succeeded OLAP queries per hour. The bottom part of Figure 4.6 shows the average abort rate, measured for same the set of experiments. I aggregated the aborted and retried transactions involving serialization failure that resulted from the CH-benCHmark log and summarized the rate as the quotient of OLTP transactions and OLAP

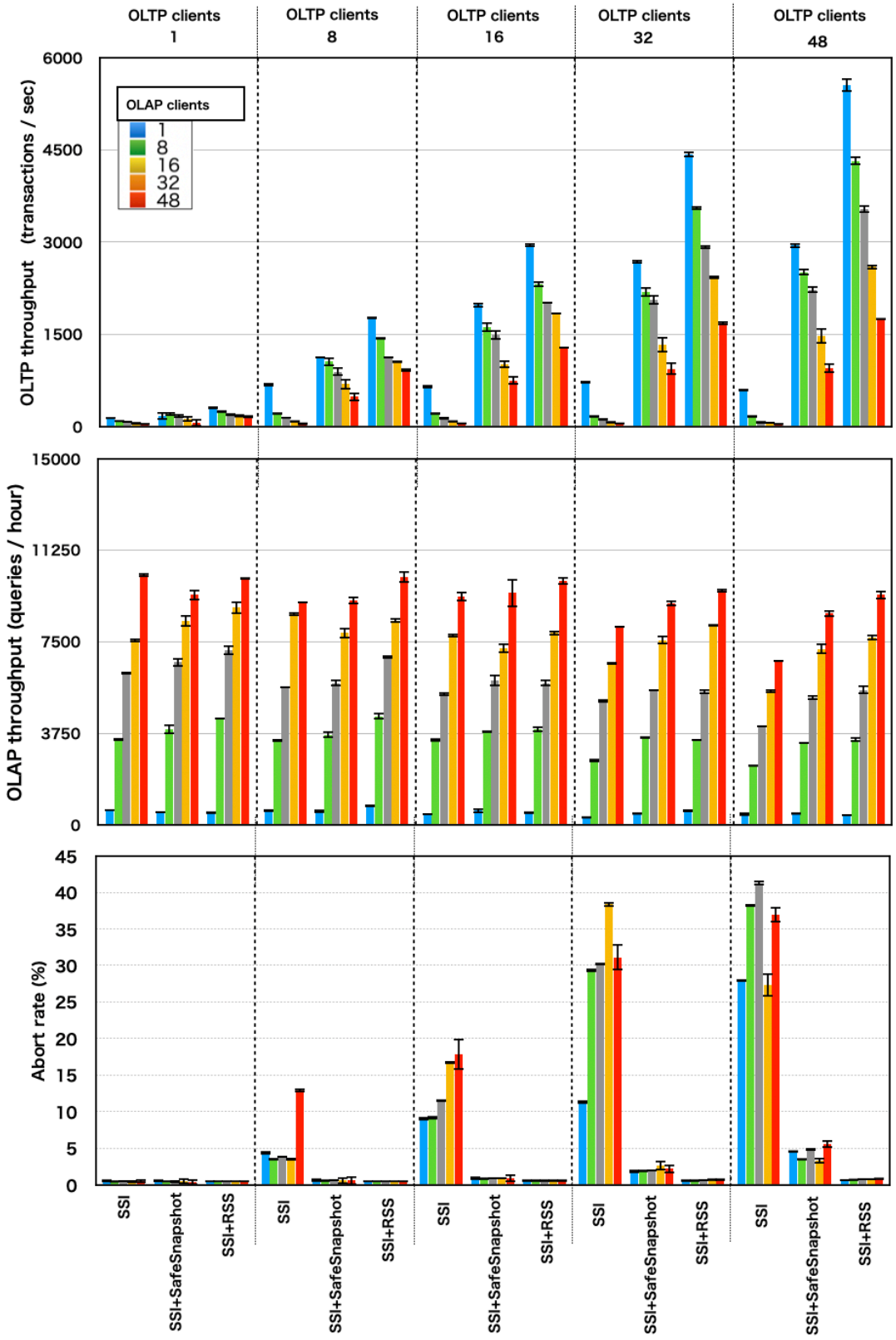

Figure 4.6: OLTP/OLAP throughput and abort rate on a unified storage architecture

queries.

HTAP systems are expected to support OLAP without seriously degrading the OLTP performance. My experiments showed decreasing OLTP throughput as the number of OLAP clients increased. As shown in Figure 4.6, this implies a serializability impact from the abort rate that exhibits a similar tendency, rather than resource competition between OLTP and OLAP. However, OLAP throughput did not change significantly (see the middle part of Figure 4.6), even though the number of OLTP clients increased. In particular, the SSI abort rate for 48 OLTP clients was about 30% when abort transactions occurred, even with the participation of one of the OLAP clients. Under the CH-benCHmark, OLAP read-only transactions are not aborted in exchange for aborting OLTP write transactions by PostgreSQL's SSI. This is because dangerous structures tend to capture the long-running scan-heavy OLAP transactions as the first reader transaction forming two successive rw dependencies, in contrast to the short-running OLTP write transactions. OLTP throughput is decreased by writer-aborts from read-only transaction participation. In addition, SSI is difficult for decoupled storage HTAP systems that cannot efficiently execute such writer-aborts, as described in Section 3.1.3.

Consequently, SSI+SafeSnapshots applying the read-only optimized SSI of PostgreSQL [PG12] could perform better than using SSI for OLTP and OLAP throughput. The overall abort rate under SSI+SafeSnapshots was lower than for SSI, because read-only transactions could avoid such writer-aborts and rw dependency checks in the SSI validation (taking advantage of reducing the cost of the SIREAD Lock [PG12]) by reading snapshots that ensured serializability beforehand. In addition, the OLAP throughput was less affected by the reader-wait problem of read-only deferrable transactions having to wait at the time the safe snapshots are constructed. It was considered that TPC-C workloads could skillfully frame transactions as benchmarks.

As shown in the middle part of Figure 4.6, the OLAP throughput of SSI+RSS was about 13% higher on average than for SSI+SafeSnapshots, across the entire range of 1 to 48 OLAP clients. To realize an efficiently serializable HTAP application, RSS can ensure serializability without reader- or writer-aborts by abopting read-only transaction participation in methods that construct serializable snapshots beforehand (wait-free snapshot read). However, the RSS OLAP throughput was similar to that for SSI+SafeSnapshots. Compared with SafeSnapshots construction methods, the RSS construction method described in Section 4.1.1 can execute OLAP read-only transactions, ensuring serializability while executing concurrent long-running write transactions on the OLTP side. For the CH-benCHmark, which has the workload features of TPC-C such as short-running write transactions, write transactions started before the read-only deferrable transactions of SSI+SafeSnapshots could finish within a short time. Although

SSI+RSS collects rw conflicts and start/end aspects of its extended information, the snapshot construction overhead is considered to be lower than for SSI+SafeSnapshots. OLAP read-only transactions under RSS could have executed no-wait operations but the reader-waits in SafeSnapshots would affect the OLTP performance, being caused by both the long lifetime involving reader-waits and the version traversals of PostgreSQL derived from old-version preservation intervals. The SSI+RSS overhead was lower than the SSI+SafeSnapshots overhead with respect to abort rate when the number of OLTP clients was increased. The SSI+RSS OLTP throughput was about 45% higher than that of SSI+SafeSnapshots in my experiments. SSI+RSS had an improved OLTP performance despite the overheads involved in collecting extended information in the OLTP-side components.

4.3.2 RSS handling costs for read-only replicas

I now describe an experiment, designed to measure RSS construction overheads for a multinode (decoupled) storage HTAP system. I examined the OLTP and OLAP throughput for the CH-benCHmark in the same way as for the unified storage system experiments. I evaluated my system, which could achieve serializability, in a comparison with SSI+SI (which could not achieve serializability for its OLAP replicas because of read-only anomalies) to investigate the overheads involved in its serializability assurance. In the same way as for the unified storage experiments, the OLTP/OLAP throughputs were averaged over 10 runs. The execution times for both SSI+SI and SSI+RSS are shown in Figure 4.7, with the lower part of the figure giving the abort rate.

I consider that the implementation restrictions were small when applying my model to HTAP systems because the OLTP/OLAP performance for RSS compared favorably with SI, even for my naïve implementations. The results show that the overall OLTP throughput fell when the number of OLAP clients increased. On the OLTP side, the SSI+SI throughput was always better than that of SSI+RSS. Although the OLAP of SSI+RSS has wait-free reads (by constructing RSS), the performance is also almost 10% lower than that of SSI+SI. I consider that the OLTP degradation in SSI+SI and SSI+RSS was caused by traversing old versions in the OLTP node. This required delaying the HOT mechanism of PostgreSQL from using the OLAP-side's hot_standby_feedback. I set the hot_standby_feedback and replication slot that PostgreSQL offers for version preservation of read-only replicas to enable correct evaluation of SSI+SI because an OLAP query can result in errors when trying to read deleted versions in the WAL that conflict with deletes. I used the same settings for SSI+RSS. In particular, by reading the versioned tuples in the direction of oldest to newest in the transaction processing

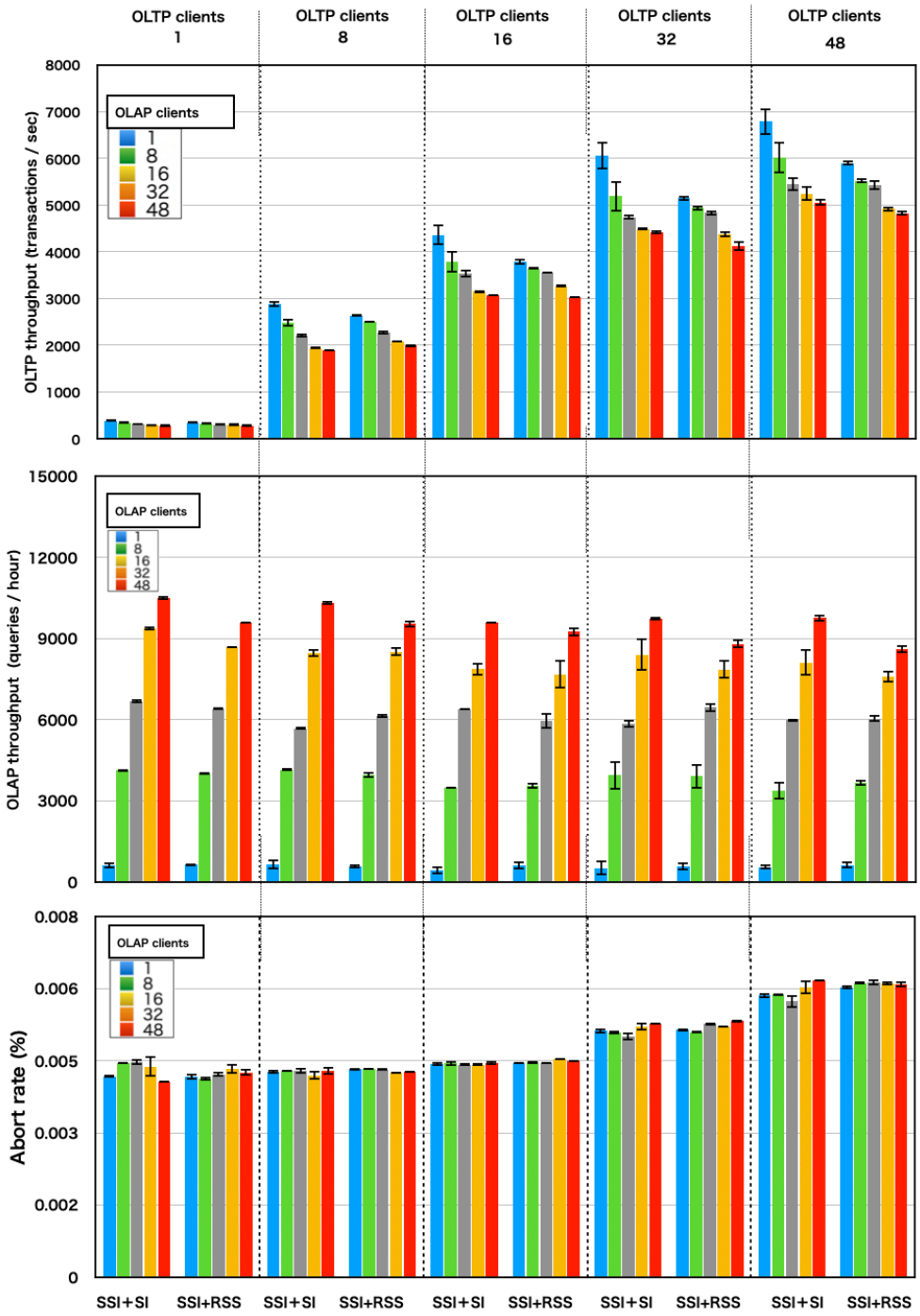

Figure 4.7: OLTP/OLAP throughput and the abort rate for a decoupled storage HTAP architecture.

of PostgreSQL, OLAP clients would cause OLTP performance deterioration. Furthermore, the OLTP side of SSI+RSS forces each write transaction to log extended information, which again caused OLTP/OLAP performance deterioration.

These factors demonstrate that the serialization cost of OLAP read-only transactions would exceed those for nonserializable analysis of SI in real time, as indicated by the maximum OLTP throughput being 17% lower than for SSI+SI during experiments run on 48 OLTP clients and 1 OLAP client. The middle part of Figure 4.7 shows that the maximum OLAP throughput is 12% lower than for SSI+SI. However, this was caused by logical decoding overheads, as described below. The lower part of the figure gives the abort rate for both OLTP transactions and OLAP queries. Compared with the unified storage experiments, the abort rate was low because the OLTP node certifies only OLTP transactions issued from TPC-C workloads under SSI. Therefore, SSI+SI has the possibility of read-only anomalies contributing to the abort rate in unified storage experiments because SI-based OLAP transactions simply read committed data available to the OLAP node.

4.3.3 Data delay time for data committed in an OLTP node to become available to an OLAP node

I now consider the delay time for RSS construction and the differences between the OLAP-node current time and the latest committed time in the OLTP node. The most-recent committed timestamp is attached to a tuple in RSS and SI. RSS properties involve finding versions (views/snapshots) that ensure serializability in OLAP-side components without the aborts/waits of OLTP transactions, but the snapshots may possibly contain somewhat stale data. Moreover, there are implementation overheads because of physical restrictions required for RSS construction: namely, those that come from checking start/commit/abort information and dependencies in read-only replicas after log-shipping data propagation.

I therefore added some parameters to the decoupled HTAP environment that would enable PostgreSQL functions to obtain commit timestamps. The performances I observed in these experiments were little changed from those in the previous experiments. PostgreSQL prepares snapshot data in terms of the visibility of versioned tuples and records the committed timestamp of a transaction by writing tuples in the header information for OLTP-side components. I enabled the track_commit_time flag and measured the time of newly taken snapshot data under SI and RSS in an external module (the RSS construction invoker), which also executed the PostgreSQL UDF's pg_xact_commit_timestamp in the OLAP node, thereby placing place the most-recent committed timestamps from a transaction ID into the taken snapshot data for SI and

RSS. To be able to use the function pg_xact_commit_timestamp, I created an RSS construction UDF to return the transaction ID having the most-recent committed timestamp in the RSS. The processes of logical decoding and RSS construction, prepared as a UDF, are needed by the time an RSS is available on the OLAP node. Therefore, SI snapshots taken at the time of the logical decoding are completed before the RSS construction UDFs start. RSSs were taken at the time the logical decoding and RSS construction UDFs ended. In addition, the total time to decode the extended log (start/end/dependency information) for RSS, which can classify the RSS log via the WAL identifier that PostgreSQL offers as a log prefix, was measured while the logical decoding was being executed. I used 48 OLTP/OLAP clients with the CH-benCHmark SF100 for the decoupled storage HTAP environment of SSI+RSS described above. In these experiments, the test duration was 30 minutes (1,800 seconds) and the warmup duration was 60 seconds after the initial data loading (about 20 minutes) of SF100 was completed. The total run time was about 50 minutes. SSI+SI was completed after data loading, as shown above, but experiments lasting 1,800 seconds gave the same OLTP performance under both SSI+SI and SSI+RSS, offering 3,200∼3,500 transactions per second.

Figure 4.8 shows the differences between the latest committed timestamp in the OLTP node and the current timestamp of the snapshots taken for SI in the OLAP node, with the time of SI being described as the "Streaming replication overhead." This streaming replication overhead is the delay between the most-recent committed time in SI and the time the snapshot data is available to the OLAP node. The processing time of PostgreSQL's logical-decoding function is described as the "Logical decoding overhead." Here, "RSS log decoding" is the time required to decode the start/end/dependency information in the logical decoding. The delay in RSSs between the most-recent committed time in the RSS and the time the snapshot data is available to the OLAP node is described as the "RSS establishment overhead." The RSS establishment overhead shown in the figure refers to the delay between the time that the RSS is available and the current time snapshot taken in the OLAP node, which reads snapshot data via a PostgreSQL function. Table 4.1 shows the breakdown of these times for Logical decoding overheads and RSS establishment overheads. RSS construction implies that my implementation will incur overheads in creating the OLTP start/end timestamp order and the OLTP transaction-dependency graph.

I consider this to be a more realistic assessment of the delay in reading the latest data (while guaranteeing serializability and while OLTP is executing transactions concurrently) than delays caused by nighttime batch propagation. Figure 4.8 and Table 4.1 show an average delay time

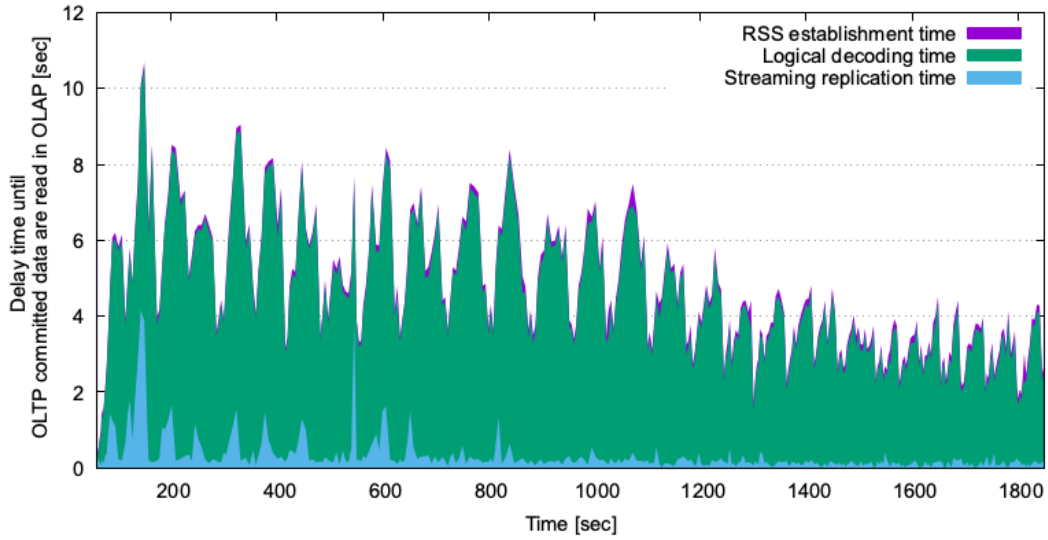

Figure 4.8: Overhead delay times between reading the latest updated data in the OLAP node and being committed in the OLTP node.

Table 4.1: Breakdown of overheads causing data delay.

Graph legend	Measurement items	Average (sec)	Minimum (sec)	Maximum (sec)
Streaming replication time (SI)	Log-shipping delay	0.307	0.009	4.175
Logical decoding time	PostgreSQL logical decoding	3.667	0.074	8.233
	RSS log decoding	0.281	0.006	2.057
RSS establishment time	RSS construction	0.129	0.039	0.654
	Serializability assurance delay	0.029	0.004	0.259
	Total	4.414	0.374	10.689

of about 4 seconds, up to about 11 seconds when using RSS. Note that new SI snapshots can be taken even while an RSS is in the process of being constructed. This means that, in the most extreme case of new SI snapshots being taken with no delay time through processes such as logical decoding, they could be up to 6 seconds faster than using RSSs. The delay for the snapshots in SI and RSS is derived from the difference between the streaming replication overhead and the total in Table 4.1. However, from this table, the logical decoding was the main factor affecting data delay in RSSs. Therefore, data delay could be most-effectively reduced by improving the logical-decoding performance. However, PostgreSQL is now well established and has become widely popular for DBMSs. In my naïve implementations, where most of the logical decoding overhead involves data delay but where there is little RSS log decoding, the data freshness would be improved if PostgreSQL's logical decoding were to be improved. Again, the RSS construction time was less significant than that for the logical decoding process. The measured values for the Serializability assurance delay confirm that the theoretical delay in my model of RSS scheduling serializability should be small for CH-benCHmark workloads.

I gave an estimate of RSS's memory consumption with the same setting as the measurement of data freshness for examining the impact of RSS construction. This evaluation was executed in the SSI+RSS environment of decoupled storage system. I used the CH-BenCHmark setting of scale factor 100 and the configuration of each OLTP and OLAP clients were 48. The result shows In Table 4.2 that the value of average, minimum and maximum as of the time executing garbage collection that new RSSs had been created at average interval of about 4.4 seconds during 1800 seconds. The overall memory consumption of database was about 10–15 GB during this evaluation.

Table 4.2: Memory consumption.

Measurement items	Average (KB)	Minimum (KB)	Maximum (KB)
DSG info.	13.348	0.256	429.056
RSS Info.	512.654	29.832	1486.344
History Info.	344.550	32.584	1002.712

In the benchmark test, the overall memory consumption of RSS construction was about 3,000KB and I consider that the memory consumption of RSS construction algorithm is few. "DSG info." in the table represents the information of an rw dependency edges needed to manage DSG and they mainly consists of two transaction IDs and the direction of edges. The

average memory consumption of the "DSG info." was about 13KB because occurrence frequency of an rw dependency in CH-BenCHmark was also low. HTAP systems generally uses CH-BenCHmark for evaluations. Read-only anomaly does not occur by reading RSS, hence RSS can be considered to avail for a variety of HTAP systems if the memory consumption of RSS construction is low.

Memory consumption of RSS construction was not dominant in comparison with the overall memory consumption (about 15GB) of PostgreSQL during 1800 seconds. "RSS info." in the table represents the information of newly created RSSs, and the information consists of transaction IDs, the numbers of dependency edges, its outgoing dependency edge from the transaction ID, etc. The average memory consumption was 512 KB and the maximum was 1486 KB. In addition, "History info." in the table consists of transaction IDs and the timestamps of start and commit/abort operations. The average of the memory consumption was 344 KB and the maximum was 1002 KB. I consider that these results was obtained due to the narrowness of the search range as the model of RSS, moreover the SSI-based optimization is expected to be more effective in other benchmarks of real world than benchmarks derived from TPC-C. The higher the OLTP load is, the better the performance becomes if a high-performance single-node system uses RSS for the serializable execution of read-only transactions instead of using other methods. Existing methods must include aborts and waits to be serializable even if read-only. Aborts and waits increase in line with the increase in concurrent transactions. For real-world workloads, read-only transactions can read RSS wait-free if long-lifetime-write transactions are executed.

I could confirm the usefulness of RSS because the memory consumption was small, the serializability was achieved, the OLTP/OLAP performances of an unified architecture improved, and the system of a decoupled architecture did have little impact to the OLTP, OLAP, and data-freshness performances as HTAP. I consider that the OLTP performance of a high-performance RSS-based system would be only about 15% worse than of the SI-based system. My implementation of PostgreSQL must find versioned tuples from oldest to newest, and that impacted the OLTP performance, as shown in my evaluation of the decoupled architecture (upper part of Fig. 7). The experiments demonstrate an OLTP performance degradation of about 15% as the worst case when the load from OLAP clients is low (the experimental setup, where OLAP client is 1 and OLTP clients are 48, showed the most adverse effects of my method). The overheads with regard to the RSS construction is inexpensive for HTAP systems, if read-only transactions will join, additional aborts/waits for guaranteeing serializability can no longer be needed.

4.3.4 OLTP and OLAP performances under serializable multinode systems

I compare the serializability impact on the multinode decoupled storage system with a serializable HTAP system because the implementation of PostgreSQL SafeSnapshots was not prepared in the read-only replica. I used CockroachDB [TSM+20] as a representative HTAP system that can achieve serializability. As discussed in Chapters 1 and 2, if a multinode environment tries to achieve serializability, ensuring serializability will cause the performance to deteriorate. However, CockroachDB can firmly achieve serializability. Thus, I compare PostgreSQL-SSI+RSS with CockroachDB v23.2.0 under my multinode environment, each of which has one OLTP-side and one OLAP-side system. I ran CHBenCHmark of SF100 issued by OLTPBench under a machine comprising an Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz with 16 physical cores, 1TB RAM, and 512GB SSD running Ubuntu 20.04.6. That same machine was prepared one more as an OLAP-side system. OLTP transactions were executed on the OLTP-side system and read-only OLAP transactions were on the OLAP-side system. In order to show serializability impacts when OLAP clients increased, I adopted eight OLTP clients while changing the number of OLAP clients to one, eight, 16, and 32 per run. Figure 4.9 shows the average of the throughputs of six runs with the standard error.

As seen in Fig. 4.9, PostgreSQL-SSI+RSS resulted in higher OLTP performance than CockroachDB. In CHBenchmark, the participation of OLAP queries causes the number of aborts/waits to ensure serializability, so CockroachDB is thought to be greatly affected by making write transactions wait. Since the OLTP side of PostgreSQL is SSI, serializability can be achieved by separating OLAP using RSS and the probability of detecting dangerous structures of SSI and aborting is also small on the OLTP side. It has already been shown that TPC-C alone does not cause a DSG cycle in [FLO+05], and even in CHBenCHmark, there is a high possibility that even an SSI dangerous structure will not occur. From previous experiments, it is clear that CHBenCHmark derived from the TPC-C benchmark is less likely to cause rw-conflicts just in the OLTP side of a multinode environment. This is because the abort rate in the single-node environment of SSI in Fig. 4.6 has decreased (as shown in the abort rate in Fig. 4.7) due to the separation of OLAP queries in the multinode environment and the memory consumption of DSG info. Just by participating in the OLAP side that reads a huge data set, rw-conflicts are more likely to occur in CHBenchmark, and not only dangerous structures of SSI but also DSG cycles are more likely to occur. However, RSS can achieve serializability without validation of read-only transactions, thereby reducing the communication cost between OLTP and OLAP. Specifically, the serializability can be maintained by unidirectional communication

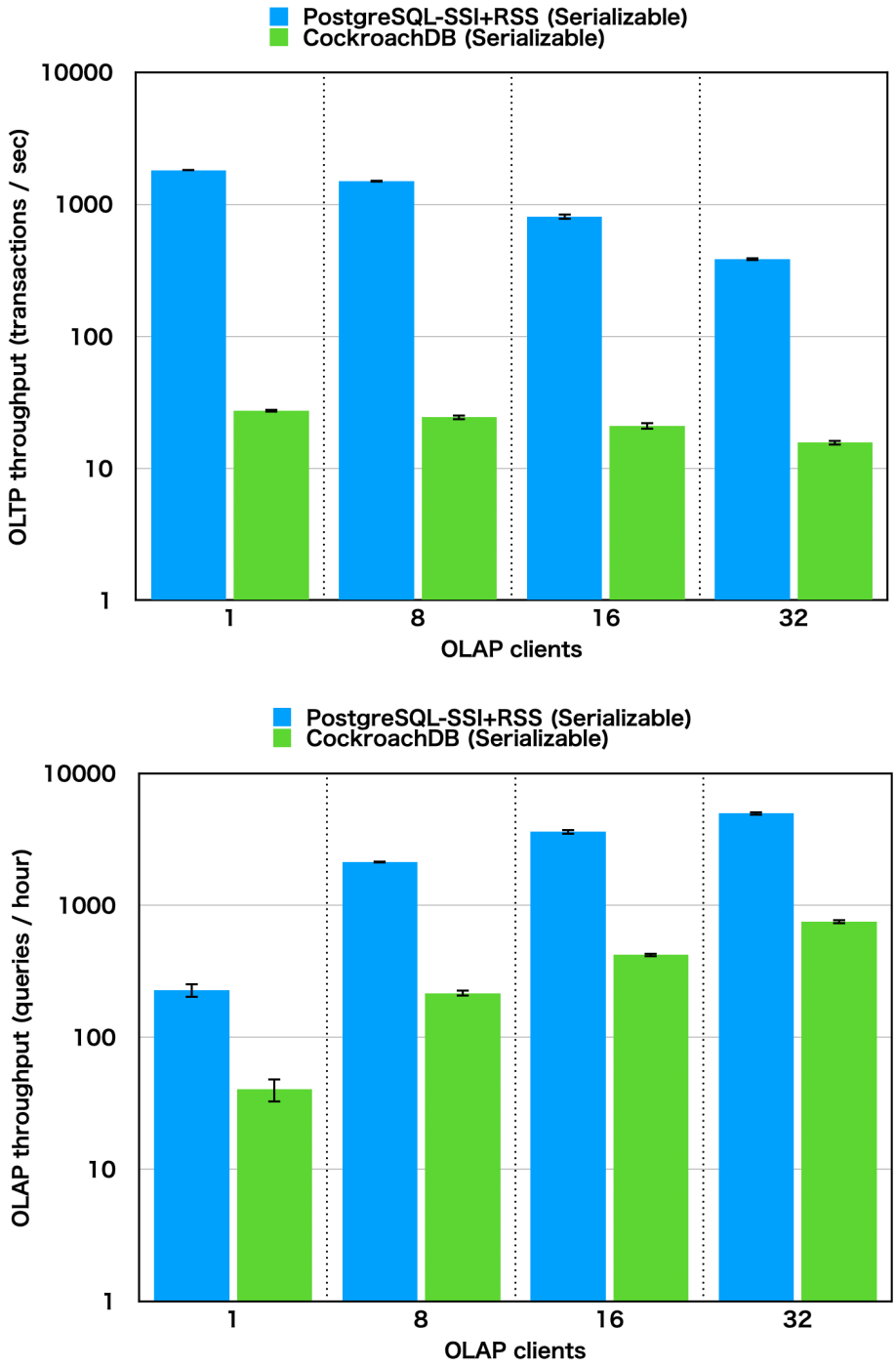

Figure 4.9: Comparison of PostgreSQL-SSI+RSS and CockroachDB in OLTP and OLAP performance under CHBenchmark.

from OLTP to OLAP without making the OLTP side check the OLAP-side information such as rw-conflicts.

CockroachDB's serializability approach is to disable over-writes by blocking writes on a read-set of OLAP queries. This is because read-only OLAP transactions are heavy and tend to have a long lifetime. As discussed in the Related Works chapter, this approach is similar to Hyper-Fork's serial execution with regard to serializability. These approaches match the conflict order and commit order of transactions in order to guarantee the serializability. In other words, the approach of CockroachDB does not cause read-write conflicts making the condition of DSG cycle, and therefore it achieves serializability. When heavy read-only OLAP transactions start before short OLTP write transactions, CockroachDB will block the OLTP write transactions if they try to commit before the OLAP transactions commit. If OLAP workloads are light, CockroachDB will make OLAP re-read data in order not to create read-write conflicts. The blocking of writes frequently occurs because OLAP reads are heavy by CHBenchmark. Therefore, the performance deterioration in order to ensure serializability is increased, as shown in Fig. 4.9.

Chapter 5

Data conversion from row to column

With RSS eliminating the need to take a huge lock on the OLAP side of HTAP sytems, and on the premise that serializability is guaranteed in storing multiversion data, how much performance improvement and implementation flexibility will OLAP be able to show? Although the previous chapter resolved theoretical problems which OLTP/OLAP/data freshness performances get worse when physically processing ability are high, the pseudo HTAP systems had processed data of the OLAP-side components by row-oritented. This chapter introduces a research to address implementational problems on enhancing OLAP performance of DBMS handling HTAP workloads. Generally speaking of HTAP, if an OLAP-side engine execute queries by using row-oriented data, the OLAP throughput will be slower. To enhance OLAP performance, many HTAP systems have addressed data conversion from row-oriented to column-oriented. The aim of this study shows an implementational approach of an HTAP system by evaluating the method of conversing the updates, which have been the physical constraint on the hybrid system of OLTP and OLAP.

There is also the issue of different access units for OLTP and OLAP between the data conversion from row to column and guaranteeing serializability. As I explained in the previous chapter, HTAP systems would want the row-by-row updates to be reflected column-wise data immediately, but it has high potential that serializable isolation cannot guarantee even if the systems read the latest updated version. Since OLTP manages updates on a row-by-row basis, its theoretical framework is also based on row units. In fact, even in the previous chapter, data objects were formulated in terms of tuples (rows). HTAP systems immediately store data column-oriented in DRAM or non-volatile storage to improve the performance of OLAP. Hyper-MVOCC uses a column-oriented locking protocol such as predicate locking to detect overwrites as an optimistic concurrency control (OCC). However, locking methods degrade

the OLTP performance by the large read set that OLAP transactions read.

This study assumes that the data contained in RSS in the temporary storage area on RAM is sequentially converted. Considering the assurance of serializability and the application of RSS, the architecture having temporary storage transforming data from row to column such as conventional column-oriented DBMS would be presumed. Although RSS does not require to take locking to data of OLAP side, the data object versions of OLTP side must be temporarily stored on DRAM, etc. Moreover, such architecture has factors that prevent the data conversion task from satisfying the demand of HTAP applications regarding OLTP/OLAP/Data-freshness performances. Conventional DBMSs based on a column-oriented storage tend to be used to OLAP system that accumulates updated data on a DRAM and bulk-loads it into a disk for compression effects. This study introduces a mechanism used by the tuple identifier (TiD) as a data object unit based on MVCC. However, the column-based DBMS has problems which increases disk I/O and log spaces in the disk for performing real-time OLTP and large data analysis. Therefore, I propose methods assumed to use non-volatile memories (SCMs: storage class memories) and updating the column-oriented data on SCMs. In this chapter, I evaluate such the write back cache methods as a criteria based on LRU (Least Recently Used) and a block having a number updated to data of the block on the columnar storage, thereby aiming the improvement of system performances.

5.1 A mechanism presuming RSS to enable column-oriented optimizations

SCMs are a type of non-volatile RAM with faster access than an SSD; therefore, I consider a mechanism has methods that enable to use more cache domains on DRAM for handling OLAP by performing OLTP using the row-oriented storage method in SCMs. This dissertation considers data deletions and modifications for OLAP and OLTP using sparse reading. I propose a DBMS architecture SCMAT that efficiently uses the SCMs for OLAP in real time by creating an update index that updates columnar buffer data on the DRAM and creates a materialized view.

In a column-oriented DBMS, while row data is stored in memory, row data is stored as files on disk in units of transactions. Conventional column-oriented DBMS that adopts a delta buffer for performing OLAP and OLTP must read data from the two fields on the disk and on the DRAM updated by INSERT, DELETE, and UPATE operations. The column-oriented file format is considered to be inefficient because it generates I/O for secondary storage in units of

columns for OLTP data update processing that is generally executed in units of rows. Therefore, HTAP application workloads cannot be handled efficiently. In recent years, practical use of non-volatile memory such as MRAM which is kinds of SCM, which has the same access speed as DRAM, is considered to be imminent, and SCM is possible to improve DBMS performance by effectively utilizing the latest devices. Since it consider updating the data on the DRAM to be faster for data analysis via OLAP, the propose architecture performs OLAP reflecting UPDATE operations without reading data from the disk in OLTP. UPDATE operations for the column-oriented storage are not faster than the row-oriented storage because the row of a table is physically required to separate and replicate into columns and projections. Therefore, I propose an index that conducts UPDATE operations on the column in the RAM, for performing UPDATE operations to replace the column values in real-time OLAP. The methods aim to show the use of SCM and reduce disk I/O to improve HTAP performance regarding OLTP/OLAP/data-freshness.

5.1.1 TiD-based column-oriented update index for improving OLTP performance

The update index is based on a hash table created by using tuple identifier TiD (or object id) of a row-oriented RDBMS. OLTP modifications uniquely modify the columns of a tuple to column-oriented data on DRAM by a TiD in the hash table. Although the column-oriented storage cannot fast perform UPDATE or DELETE operations than a row-oriented storage due to sort each columns by the column value order, the update index aims to fast specify a row of each column. Therefore, I propose a method to determine when updating a row update to a block storage, based on a criterion that combines both the number of data in a block derived from column-oriented storage and LRU value of row. The Index manages TiDs as a hash key for specifying the columnar data updated by OLTP on SCM, moreover it computes the LRU hit count number and logical block Identifier (LBID) corresponding to the updated rows.

In addition, it computes the number of the update rows in the column-oriented block processed by the conflict of hash table. The criteria for writing data on disk define in descending order by the average both of the number of update rows in a columnar block, with regard to whole statistics stored as metadata, and the total value of LRU in block. The index manages identifiers of OLAP-side data and OLTP-side updates in the processing components, which (1) obtain the TiD and LBID of columnar data in a disk by using the WHERE clause of the UPDATE or DELETE query, (2) store the tuple in the temporary storage on DRAM presuming a SCM by using the TiD, (3) modify the column data on the DRAM by searching for a value in the hash table using the TiD list as the key of the hash table, implemented as shown in Algo-

rithm 3, 4, 5. Disk I/O by reducing the number of blocks written back while considering cache hits to row data on the SCM so that the tuple reconstruction cost due to column-by-column reading from the disk during OLTP execution can be reduced as much as possible.

I reports a preliminary experiments evaluating the write back block reduced by the proposed methods. Using the experimental environment and TPC-C benchmarks described in Section 5.2, I evaluated how much updated rows were included when the block was returned as follows. The setup of experiments ran TPC-C benchmark with and SF and clients set to 4, 14 were run for 120 seconds each in Table 5.2 and Table 5.1.

Table 5.1: Write back tuples on TPC-C benchmark (SF=4).

SCMAT with equipping a method	Number of times swapped out	Number of blocks swapped out of memory	The average number of rows in blocks written back to disk
LRU calculation per block	5965	27642	216
LRU calculation per row	978	4196	484
Calculate the average of row LRU value and updated row in block	93	4374	4107

Table 5.2: Write back tuples on TPC-C benchmark (SF=14).

SCMAT with equipping a method	Number of times swapped out	Number of blocks swapped out of memory	The average number of rows in blocks written back to disk
LRU calculation per block	15987	72081	84
LRU calculation per row	4660	20792	2456
Calculate the average of row LRU value and updated row in block	127	18900	2479

Since the number of tuples written back is a column-oriented block, there are two types: tuple ID and attribute. The computational overhead of the proposed method is compared with that of conventional column-oriented DBMS in section Section 5.2 as the whole OLTP performance. Note that the column-oriented DBMS is likely an open-source version of a commercial DBMS, stripped of various optimization options.

Algorithm 3 Update processing method on DRAM

Input: $cachedPage, LBID, colnum, TiD$

$t \Leftarrow hash(TiD)$

$t{\rightarrow}cacheflag[colnum] \Leftarrow \&idx[TiD]$

if $t{\rightarrow}delFlag$ **then**

$pageDeleteVector\ append\ idx[TiD]$

$t{\rightarrow}TiD \Leftarrow TiD$

while $(uflag \Leftarrow t{\rightarrow}UPDflag) \neq NIL$ **do**

if $uflag{\rightarrow}LBID$ is LBID **then**

$ver \Leftarrow create\ new\ version\ node$

$i \Leftarrow idx[TiD]$

$ver.TxnID \Leftarrow TxnID[i]$

$ver.data \Leftarrow colData[i]$

$verFlag[i] \Leftarrow \&ver$

$scmcol \Leftarrow scmCellRead(TiD, colnum)$

$TxnID[i] \Leftarrow scmcol.TxnID$

$colData[i] \Leftarrow scmcol.data$

$uflag \Leftarrow uflag{\rightarrow}next$

5.1.2 Data relocation/conversion for improving OLAP performance

The architecture of SCMAT uses the column-oriented storage in fast-access DRAM for performing OLAP. OLAP must process many tuples, and thus, the architecture I chose is based on reading the many tuples on DRAM, rather than on loading data from the disk. Moreover, row-oriented storage performs OLTP on SCM to handle OLTP without reading data from the disk and creates the update index in SCM to perform OLAP by modifying the column data on the DRAM. The conventional column-oriented DBMS (which OLTP requires for the updated non-volatile data to be processed) must store the data on volatile DRAM and store the log on the disk. Sush architecture must immediately store data to the row-oriented storage using non-

Algorithm 4 Delete processing method on storage

Require $currentScmTuple, updateColumnNums, delFlag$
for $col = 0$ *to* $columns - 1$ **do**
  if $t{\rightarrow}cacheflag[col] \neq NIL$ **then**
    $i \Leftarrow t{\rightarrow}cacheflag[col]$
    $ver \Leftarrow create\ new\ version\ node$
    $ver.TxnID \Leftarrow TxnID[i]$
    $ver.data \Leftarrow colData[i]$
    $verFlag[i] \Leftarrow \&ver$
    $pageDeleteVector\ append\ i$
    $LBID \Leftarrow getLBID(t{\rightarrow}TiD, col)$
    $LBIDcountLRU \Leftarrow hashLRUarr(LBID)$
    while $LBIDcountLRU \neq NIL$ **do**
      if $LBIDcountLRU.LBID\ is\ LBID$ **then**
        $LBIDcountLRU.count \Leftarrow 0$
      else
        $LBIDcountLRU \Leftarrow LBIDcountLRU.next$
    $LBIDcountLRU.LBID \Leftarrow LBID$
    $LBIDcountLRU.count + 1$

volatile SCM for OLTP. My goal is to perform OLTP and real-time OLAP using the updated data.

As I described above, for this architecture, the DRAM stores data using column-oriented storage and the SCM domain has an index for UPDATE. The update index I proposed in the previous section, which uses a hash table to specify the rows of the columns based on TiD of row-oriented storage as the keys of the hash table. The TiD represents the physical location of the row in a table on the disk; thus, I create both the column and the TiD on the DRAM by adopting RDBMS in SCM as the row-oriented storage method. The address of TiD including the column value on the DRAM is stored as the value in the hash table. There are several advantages to this method when (1) performing transactions of RDBMS, (2) possessing non-volatile data, and (3) linking the data of a row of columns together by constructing the architecture in SCM. The hash table based on the TiD of the physical position stored using RDBMS.

In addition, the column-oriented data horizontally separates the segment of 150 thousand rows by reading the sparse data of the minimum value and maximum value to improve OLAP

Algorithm 5 Update processing method on storage

```
Require currentScmTuple, updateColumnNums, delFlag
updFlag ⇐ create new columnupdate node
col ⇐ updateColumnNums[0]
updFlag.LBID ⇐
currentScmTuple.LBID[col]
UpdDRAMPage(cacheMap(updFlag.LBID), updFlag.LBID, TiD)
t→scmUpdFlag ⇐ updFlag
LBIDcountUpdate(updFlag.LBID)
if updateColumnNums→len − 1 ≧ 1 then
    while col in updateColumnNums next ≠ NIL do
        updcol ⇐ create new columnupdate node
        updcol.LBID ⇐
currentScmTuple.LBID[updateColumnNums[col]]
        UpdDRAMPage(cacheMap(updcol.LBID), updcol.LBID, TiD)
        updFlag.next ⇐ updcol
        updFlag ⇐ updcol
        LBIDcountLRU ⇐ hashLRUarr(updcol.LBID)
        while LBIDcountLRU ≠ NIL do
            if LBIDcountLRU.LBID is updcol.LBID then
                LBIDcountLRU.count + 1
            else
                LBIDcountLRU ⇐ LBIDcountLRU.next
        LBIDcountLRU.LBID ⇐ updcol.LBID
        LBIDcountLRU.count + 1
```

performance. The architecture of SCMAT creates a TiD list by using the sparse reading to identify rows using the hash table from OLAP and OLTP queries.
first, my method finds to the column data of the WHERE clause on the DRAM using segment elimination [SPP16]; my method creates a TiD list. Second, the method passes a TiD list.

In this section, I describe my aggregation method, which uses the update index. The column-oriented DBMS inefficiently acquires several rows from large scale data, relative to using a B-tree index of RDBMS. My update index using the hash table can acquire the specific rows by identifying to the address of data on the DRAM. Additionally, I create a materialized view to efficiently process the complex OLAP using the queries and data for DWH. In the case that few columns and many rows are updated, my methods execute following steps; (1) obtain TiD lists from the column data on the DRAM based on the WHERE clause of the OLAP query, (2) Search and aggregate the column data based on the SELECT clause of the OLAP query by using the TiD list as the key of the hash table, and (3) create a materialized view of the row data searched in step (2). My methods use column-oriented searching based on minimum and maximum values from the buffer on the DRAM and use row-oriented searching based on the hash table with TiD. In addition, my methods create a materialized view on DRAM for OLAP from the column buffer used in the OLAP query In the case of processing query #1 of TPC-H, my methods create a materialized view using the column value in the GROUP BY clause as sort key values. Step (3) creates the materialized view of a row for each sort key value (l_returnflag, l_linestatus).

```
SELECT l_returnflag, l_linestatus,
...
GROUP BY l_returnflag,  l_linestatus
... ;                      (a portion of #1 TPC-H)
```

5.2 Evaluation

In this section, I evaluate my DBMS architecture using DRAM instead of using SCM, because SCM such as MRAM still not available in computing systems. Therefore, I used the functions of row-oriented DBMS on DRAM and create the TiD-based column update index on the DRAM to modify the column-oriented values by using a hash table.

Experiments were performed on Amazon Elastic Compute Cloud (Amazon EC2) of an r3.4xlarge[1] instance type. The instance was run on Red Hat Enterprise Linux Server release 7.1

[1] https://aws.amazon.com/ec2/instance-types/?nc1=h_ls

(Maipo), which had 122 GiB DRAM. The CPU was defined by Amazon as an Intel(R) Xeon(R) CPU E5-2670 v2 at 2.50 GHz processor; additionally, the general purpose SSD volumes called EBS (gp2) 500 GB was used. The catalog spec of max throughput per volume is 128 MiB/s and max IOPS per volume (16 KiB I/O) is 16,000, but the Amazon service illustrates that the volume might not reach full performance. I implemented architecture SCMAT by connecting the functions both of MariaDB Columnstore 1.1 of the open source version of InfiniDB and SQLite 3.8.3. In addition, since SCM is assumed to have the same access speed as DRAM such as MRAM, the area assumed to be SCM uses DRAM, so the experimental environment was set without delay.

5.2.1 Data conversion performances under OLTP workloads

This section shows the OLTP performance of proposed method only used by update index executing data conversion from row to column. Tables 5.3 show the averages of five execution results of TPC-C benchmark (SF=4) during 300 seconds, and Table 5.4 show the average of five execution results of TPC-C (SF=4) and TPC-H (SF=100) benchmark, where queries are selected at random based on themselve specification. Figure 5.1 and 5.2 show the disk I/O statics when the benchmark (SF100) query #2 was executed concurrently during about 60 seconds, which is defined by the query lifetime from begin to end. The horizontal axis of the figures represents the elapsed time of one run.

Table 5.3: Evaluation of the proposed architecture by using TPC-C (SF=4).

Evaluated system	Transaction name	Number of committed transactions	TPS(Transactions per seconds)
MariaDB columnstore	DELIVERY	7	0.15
	NEW_ORDER	60	0.27
	ORDER_STATUS	7	4.67
	PAYMENT	55	1.81
	STOCK_LEVEL	5	6.66
SCMAT	DELIVERY	12	0.24
	NEW_ORDER	153	0.69
	ORDER_STATUS	14	7.53
	PAYMENT	132	4.83
	STOCK_LEVEL	8	11.11

From the results in the tables, it is considered that the OLTP performance was improved

Table 5.4: Performance comparison during concurrent execution of TPC-C (SF=4) and TPC-H (SF=100).

Evaluated system	Transaction name	Number of committed transactions	TPS(Transactions per seconds)
MariaDB columnstore	DELIVERY	1	0.16
	NEW_ORDER	32	0.15
	ORDER_STATUS	3	3.66
	PAYMENT	32	0.25
	STOCK_LEVEL	1	4.75
	Total	69	0.19
SCMAT	DELIVERY	8	0.65
	NEW_ORDER	107	0.34
	ORDER_STATUS	13	4.64
	PAYMENT	97	3.77
	STOCK_LEVEL	7	9.58
	Total	232	0.65

by about 90% (up to approximately 2-3 times the performance improvement) when the proposed method was adopted. In the preliminary experiment of proposed columnar index and write-back method, the number of transactions that could be executed was large, and the proposed method was likely to be affected because the effectiveness of the proposed method was simulated from the results of TPC-C benchmark execution. The results of Table 5.3 indicate that it is difficult to confirm the effect of the method proposed in this study due to the small number of transactions that could be executed. Therefore, the following diagram shows diskIO in the experiment. In order to further investigate the effect of the proposed method, the results of checking the disk I/O during simultaneous execution of TPC-H query #2 and TPC-C benchmark are shown in Figure 5.1 and Figure 5.2.

As shown in the figures, the proposed method seems to not interfere with the disk I/O because the data written to the disk is collected at once, thereby reducing whole the disk I/O overhead. When the proposed method was evaluated for the average execution time of the benchmark, no performance difference was observed between MariaDB Columnstore and SCMAT, therefore, the performance of OLAP could not be improved. Even in the comparison system, the disk I/O of OLTP can be suppressed while reading the data required by OLAP at once, so the performance of OLAP on SCMAT did not improve, it seems that OLTP per-

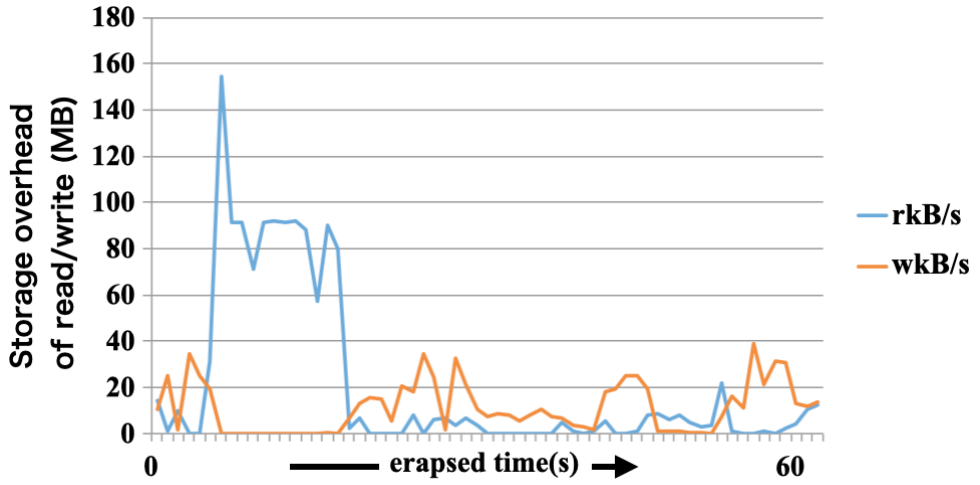

Figure 5.1: Disk I/O per second during experiments under MariaDB Columnstore.

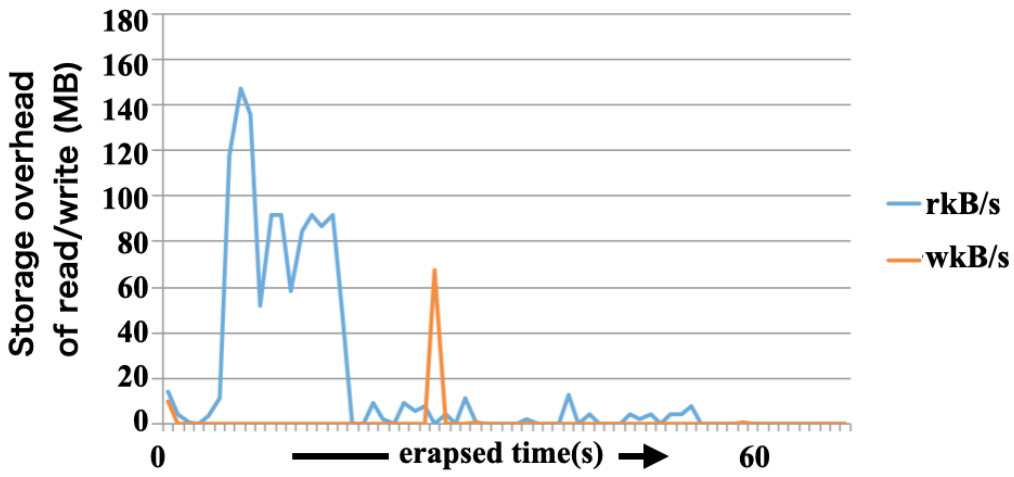

Figure 5.2: Disk I/O per second during experiments under SCMAT.

formance has improved because OLTP can execute OLAP affected at the same time. In the next section, I evaluate the combination of the proposed methods using the update index and materializations for improving OLAP performance.

5.2.2 Data conversion performances under OLAP workloads

During the course of my experiment, I generated approximately 1 GB and 10 GB of data [using scale factors of 1 (SF1) and 10 (SF10)] on a data generating tool named DBGen[2] de-

[2] http://www.tpc.org/tpch/spec/tpch2.8.0.pdf

veloped by the Transaction Processing Performance Council (TPC). I prepared TestQuery for evaluating the TiD-based column update index in UPDATE operations, where TestQuery modifies the TPC-H of the *lineitem* table for rows with a selectivity of 50%. In this evaluation of the UPDATE operation at the average of 10 runs, Table 5.5 compared against a commercial column-oriented DBMS used in data analytics widely; the system is denoted by DBMS V.

```
UPDATE lineitem
SET l_extendedprice=l_extendedprice+5
WHERE l_returnflag='N' ;              (TestQuery)
```

Table 5.5 shows the processing times in the column update index in the two-layered structure of the upper stage in which the hybrid of my implementations and SQLite functions on DRAM performed TestQuery, the middle stage for which data relocation and conversion methods performed TestQuery, and the lower stage, which shows the entire processing time of TestQuery. SQLite on memory performed fast UPDATE to the *lineitem* table of SF1 but slowly performed UPDATE operations on the *lineitem* table of SF10 (relative to the average processing time). In my modification method for the buffered column, the result is that my methods using hash tables take a few of time to modify the one-column value after specifying a row of the column because TiDs are not updated after performing UPDATE in SQLite on memory. If SQLite on memory had renewed the TiD, my TiD-based column update index would not be rapidly processed due to modify all TiD of the columns with the key of the hash table on DRAM assuming SCM. The result is that one can insert data into the empty space created by UPDATE in the case of handling a large amount of data of SF10.

Table 5.5: Update processing times of SCMAT and a commercial column-oriented DBMS (DBMS V).

Measurement items	SCMAT (SF 1)	SCMAT (SF 10)	DBMS V (SF 10)
Processing time of storing row data and conversing columnar data (sec)	10.1	105.4	
Processing time of data relocation (sec)	0.3	9.0	
Total (sec)	10.4	115.4	152.8

Figure 5.3 shows the processing times of my aggregation methods when performing TestQuery. I confirmed the aggregation times while changing the selectivity in the two pattern of

using the materialized view and the hash table. Additionally, Figure 5.4 shows the processing times of my aggregation methods in performing the query #1 of TPC-H of SF1. Figure 5.5 shows the processing times of my aggregation methods when performing the query #1 of TPC-H of SF10. The results show that the my materialization method could catch up with the processing time based on the second performance, such that the materialization does not need to consider the first processing time of creating materialized views. Therefore, my materialization method is more effective than non-materialization using only the my hash table because users of the column-oriented DBMSs perform a similar query when considering the features of the data in DWH.

The hash table could more rapidly create materialized views as selectivity decreased, such that the materialization efficiently processed the queries. However, the time required to create the materialized views was worse in the case of processing a large selectivity because the my materialization method must perform filtering to sort key values and connect the materialized views based on each sort key value.

The traditional column-oriented data storage method attracted attention because the development of column-oriented DBMS was optimized following the evolution of technology such as CPU, main memory, and disk technology (i.e., core components of a computer). Many commercial DBMSs based on column-oriented DBMS (suggested in the 2000s) have been used to analyze large-scale data. However, column-oriented DBMS is difficult to apply to big-data analysis that updates many data entries in real time because column-oriented storage performs inefficient OLTP when processing row-oriented updates. Big-data analysis requires DBMS to efficiently perform both OLTP (often with data from IoT devices) and OLAP, which immediately uses the data accumulated in the DBMS. Therefore, I enabled column-oriented DBMS to efficiently process OLTP for performing big-data analysis. In this study, I proposed a DBMS architecture by focusing on cutting-edge storages (i.e., core components of computers) such as SCM.

SCM is non-volatile RAM with faster accessing than an SSD; therefore, my proposed architecture is able to have more cache domains on the DRAM for handling OLAP by performing OLTP on row-oriented storage method in SCM. Moreover, I created a TiD-based column update index for real-time modification of the column data of the buffer on DRAM, and my experiments showed that the processing time of my TiD-based update index performs a fast UPDATE query on 25% of the data of scale factor 10 in the TPC-H benchmark relative to the processing time just after performing DBMS V.

In addition, the hash table I created is able to acquire a row of sorted column data such

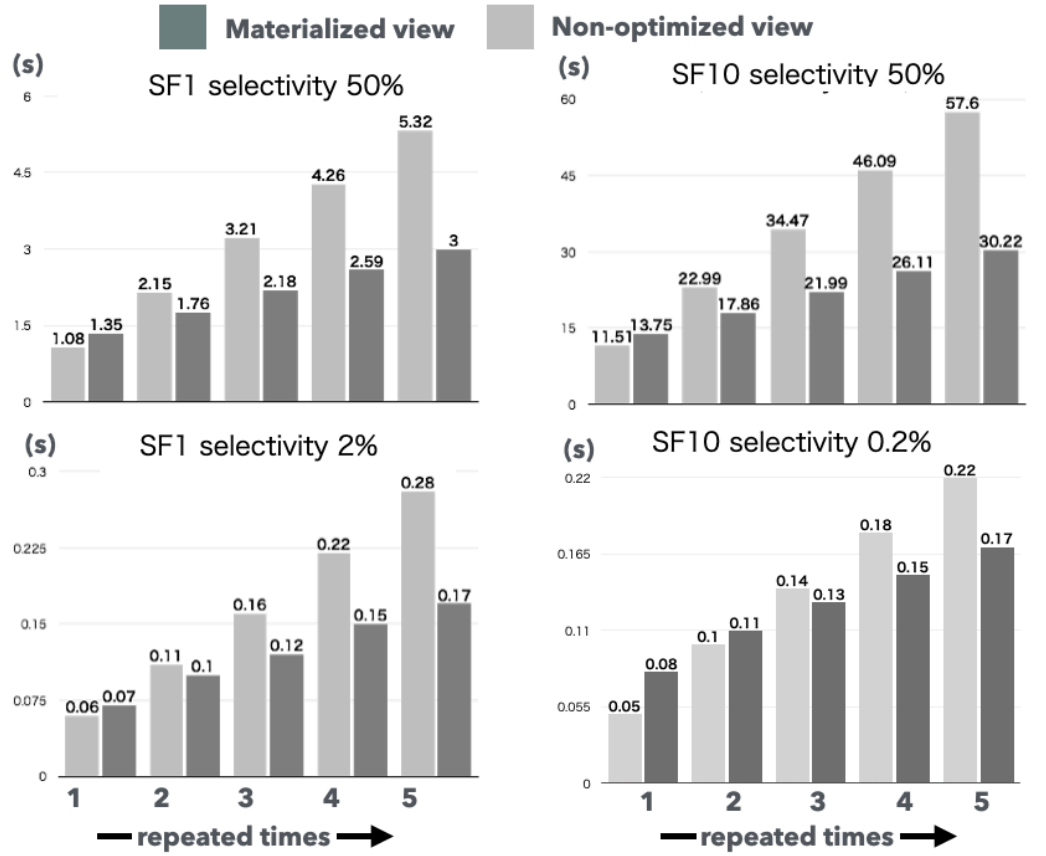

Figure 5.3: Processing times of TestQuery in the case of using materialized/non-materialized view.

that the proposed a materialization method for aggregation in OLAP can be performed. Thus, I evaluated the processing time of OLAP queries in the case of creating a materialized view. My materialization method could catch up with the processing time after the second iteration because my TiD-based column update index, using the hash table, can obtain a specific row by identifying the address of all column values on the DRAM; the column-oriented DBMS inefficiently acquires a row from a large data set. Therefore, my index is easy to use with the row-oriented method and the column-oriented storage. Moreover, the materialization is more efficient than non-materialization used only for my hash table because users of the column-oriented DBMS perform a similar OLAP query based on the features of the data in DWH.

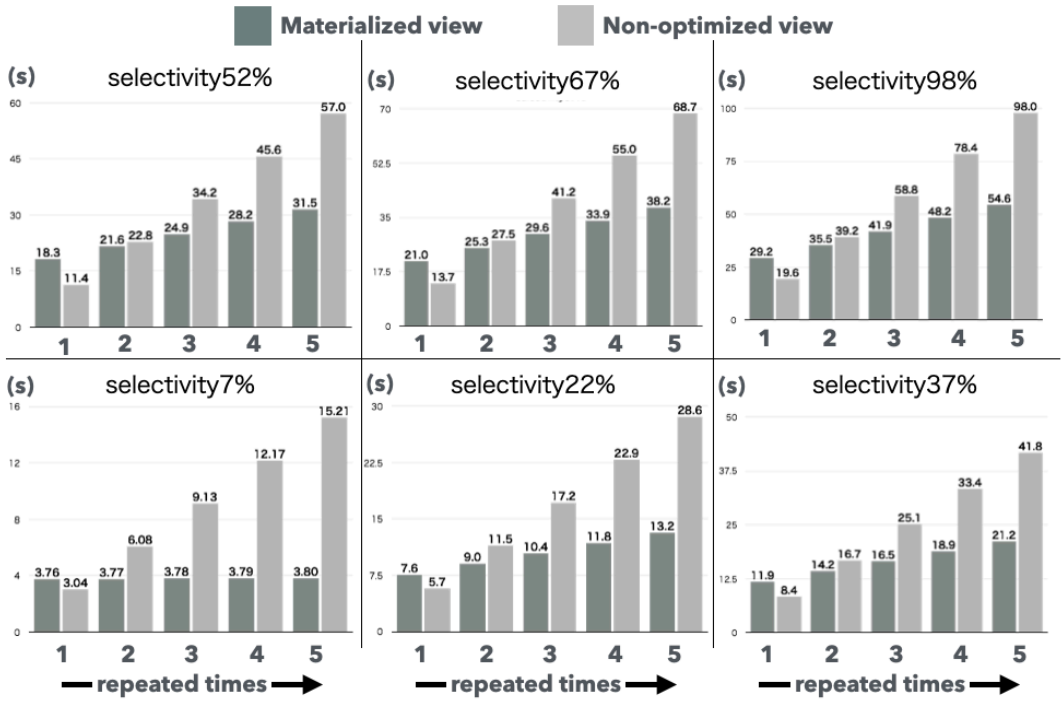

Figure 5.4: Processing times of TPC-H query #1 of SF1.

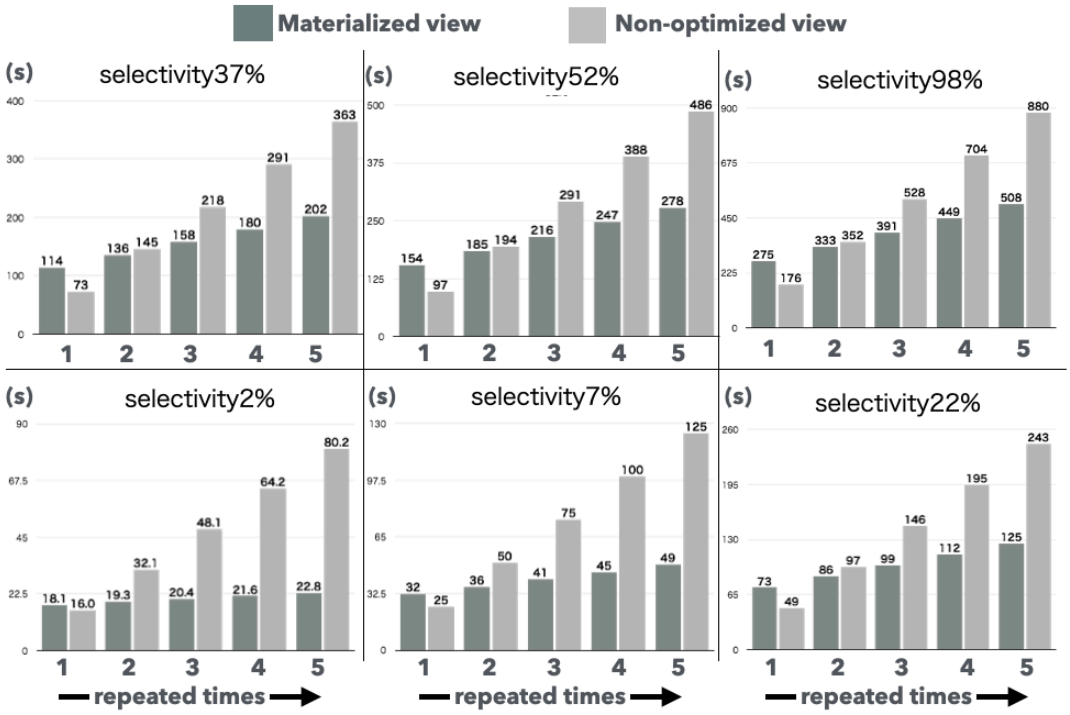

Figure 5.5: Processing times of TPC-H query #1 of SF 10.

Chapter 6

Discussion

I summarize the main points of this dissertation. Many applications that handle HTAP workloads appeared, and HTAP systems were developed and attracted attention, causing that HTAP systems is required to have sufficient performance of OLTP, OLAP and data freshness. The reason why these performances cannot be obtained considered that serializable read-only replicas cannot be easily created in the first place. Because there is only a theory that holds a theoretical problem causing aborts no matter what architecture HTAP systems is created. Therefore, OLTP, OLAP, or data-freshness performances get worse when physically processing ability (or devices) are high. First, this dissertation solved the theoretical problem, then moved on to architectural and implementation problems. The proposed systems obviously showed superiority in evaluations as HTAP (DBMS), which implies transactional, analytical, data-freshness, serializability and data-conversion performances. At any rate, the theoretical solution has a wide range of applications. It's hard to find methods to compare with because these methods never existed before, as they can make great contributions in future DBMS reserch.

These proposed methods can also be easily applied to HTAP systems developed to handle huge amounts of data such as real-time big-data analysis. Some HTAP system architectures use Spark as the OLAP-side subsystems of a HTAP system. Implementation approach will be required in the future it will be possible to flexibly optimize data structures for analytical processing by constructing RSS to OLAP-side subsystem. Such big-data-oriented systems has many future challenges, but it may become realistic and practical with improvements in physical new devices such as SCM and implementation capabilities.

Future prospect is possible to be considered to write back from OLAP side to OLTP side for user demand. There may occur an assumed use case as the example that wants to write the reports read by OLAP into DBMS. It may be considered for users to perform such case for

granted because DBMS users expect to take many use case with forgetting OLAP-side subsystems to be read-only. The case becomes difficult to meet such user demand in terms of existing theory for ensuring serializability. This is because HTAP systems must be required to revisit in the bidirectional communication from OLAP-side subsystem to OLTP-side subsystem. RSS could achieve both of consistency guarantee and performance improvement because the communication is enough from OLTP to OLAP. *Multinode HTAP systems of unified storage architecture* would be inevitable to ensure serializability in the general distributed transaction theory, therefore establishing both serializability and performance in HTAP is difficult. For example, in existing theoretical frameworks, even if write-back data from OLAP side is not read by OLTP transactions, the situation is equivalent to a situation that OLTP can not read data which reside in DBMS. I think that there is possibility of reconsideration of theoretical scalability. I anticipate that there will be many challenging elements not only for many developers existed in data management research fields but also for a few of researchers.

If DBMS application does not handle large amounts of data, the need for a columnar format is unclear. There are not many that presuppose serializability. Because almost HTAP systems are built from scratch, it is difficult for researchers to create systems such as HyPer, with comprehensively considering CC, optimization and storage storing methods from columnar. This dissertation proposed theoretical framework RSS to enhancing effect when applying HTAP. In addition, considering RSS application to an HTAP system, I introduced the implementation policy. The experimental results imply that columnar data conversion alone does not provide performance, causing the need of many optimization from columnar. Based on that, version preservation of row-oriented data is required in architecture of SCMAT such as column-oriented RDBMS.

SCMAT assumes that using NVRAM (Non-volatile RAM) is not the essence. Since NVRAM could not be used for proposed architecture SCMAT itself, DRAM was used instead, but there seems to be no problem in performance. In fact, NVRAM development seems to have slowed down and it is still a long way from the ideal NVRAM being available. Furthermore, the performance of storage has greatly improved; it is now possible to produce 10 to 20 times the throughput. The SCMAT architecture assumes fast DRAM and slow disks, but such an architecture can be considered viable even with significant increases in storage performance. This is because the advantage of columnar storage is to compress, store, and read big data efficiently, and it is ideal for such data to be non-volatile. According to experimental results, standards such as LRU tend to cause cache hits with small updates, so it may be necessary to have a structure that makes it easier to retain rows in consideration of cache hits on DRAM.

Chapter 7

Conclusion

This dissertation addresses a missing part of the MVCC aspects of read-only transactions and version selection in DBMSs. In particular, I considered that serializability, where read-only transactions frequently occur, will be needed for database applications such as HTAP. I introduced the concept of an RSS and its construction algorithm to achieve global serializability, meaning that OLAP read-only transactions can step into OLTP transaction histories by choosing previous versions. RSS can be constructed by sending only start/end/rw dependency information from the OLTP side to OLAP-side components. Moreover, RSS construction uses only the timestamps in OLTP nodes, and OLAP read-only transactions can define read targets for the OLTP transaction IDs in OLAP nodes. Even though OLTP and OLAP components are unified storage elements in distributed HTAP systems, RSSs can create exclusive snapshots for OLAP queries by collecting only integrated OLTP information.

My prototype systems applied the proposed algorithm to two types of HTAP architecture categorized: namely, unified storage HTAP systems and decoupled storage HTAP systems. I evaluated the performance of my proposed systems under OLTP and OLAP workloads. For a unified system, where serializable methods (SSI or SSI+SafeSnapshots) offered, my prototype improved the average OLTP throughput by up to 45%. In addition, the RSS construction overhead for a decoupled storage system was worse by only about 15% than that for an SSI+SI method that can cause anomalies in an OLAP replica and cannot guarantee serializability. In contrast, the SSI+RSS method can achieve serializability and the OLTP/OLAP throughput was not degraded significantly compared with other methods for both unified and decoupled architectures. The OLTP/OLAP abort rate did not increase. These results show that my approach can read snapshots that are wait-free and abort-free when read-only transactions are continuously executed. I also confirmed that the prototype of my multinode HTAP architecture out-

performed existing CockroachDB HTAP system. Moreover, my HTAP architecture presuming SCM in Chapter 5 achieved efficient update methods considering data conversion from row to column. The updating performance with the data conversion improved by up to about 3 times. The aggregation performance could be increased in about 25 %. Since there are many commercialized HTAP systems, they will have various methods but these proposed methods also could demonstrate effectiveness against such system.

In future work, I will research the detailed occurrence rate of read-only anomaly because I just reported the rate briefly in this dissertation. Moreover, I aim to improve WAL-decoding processing and develop my naïve implementations to be able to read the latest updated data. I measured the data delay time between the latest committed time at an OLTP node and the current time of taking snapshots using SI and RSS on the OLAP side. Although the time for the RSS construction process was less significant than the logical-decoding time, it was the main factor affecting data delay in RSSs. The average delay time was about 4 seconds, meaning that data freshness is virtually guaranteed for that latest available data read by users. RSS usage can apply to unified and decoupled HTAP architectures while guaranteeing the serializability of OLAP and while OLTP transactions are being executed concurrently. Such systems will incur only small delays when compared with data propagation delays associated with nighttime batch processing. In future work, I aim to handle workloads that include both OLTP and OLAP within a single transaction for execution using a single-engine architecture. In addition, I aim to study the scale-out architecture and tiering capabilities of multinode servers for RSS-based performance enhancement, based on the idea that RSS can reasonably add serializable read-only replica nodes without requiring consensus or synchronous-commit algorithms.

Acknowledgments

This dissertation would never have been possible without help and support of many people.

I would like to thank my boss Prof. Haruo Yokota (横田 治夫) for his continuing support.

I would also like to express my appreciation to my respectful teachers Takashi Kambayashi (神林 飛志), Suguru Arakawa (荒川 傑), Ryoji Kurosawa (黒澤 亮二) at Nautilus Technologies, Inc. for their kindness, patience, understanding, continuing support, and encouragement during my long long doctoral study. Without their encouraging to me, this dissertation would never been completed because it was lots of inspiration and motivation from beginning to the end. Suguru Arakawa and Ryoji Kurosawa have been extraordinarily tolerant and supportive for a very long time. I apologized for the inconvenience. My heartfelt appreciation goes to Suguru Arakawa and Ryoji Kurosawa for teaching theory, model, highly specialized technical contents, and etc. in the fields of computer science or math.

I would like thank my technical adviser Dr. Satoshi Hikida (引田 諭之) at Scalar, Inc. for his continuing support, understanding, and encouragement.

I would like to appreciate my supervisor Prof. Jun Miyazaki (宮崎 純) at Tokyo Institute of Technology, for his patience, understanding, and encouragement. I respect his great attitude as a researcher/teacher on academia; for example, he extremely contributed to my journal paper, so that I have thought he is appropriate as my co-author, but he didn't want my suggestions for that because of participation in the middle of a project.

I would like to express my appreciation to my parents Sumio (澄夫) and Narumi (成美) and my big brother Atsunori (敦升).

Bibliography

[ADHS01] Anastassia Ailamaki, David J DeWitt, Mark D Hill, and Marios Skounakis. Weaving Relations for Cache Performance. In *Proceedings of the 27th International Conference on Very Large Data Bases*, pages 169–180. VLDB Endowment, June 2001.

[AKPA17] Raja Appuswamy, Manos Karpathiotakis, Danica Porobic, and Anastasia Ailamaki. The case for heterogeneous htap. In *8th Biennial Conference on Innovative Data Systems Research*, number CONF, 2017.

[AKW+13] Dmytro Apalkov, Alexey Khvalkovskiy, Steven Watts, Vladimir Nikitin, Xueti Tang, Daniel Lottis, Kiseok Moon, Xiao Luo, Eugene Chen, Adrian Ong, et al. Spin-transfer torque magnetic random access memory (stt-mram). *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 9(2):13, 2013.

[ALO00] A. Adya, B. Liskov, and P. O'Neil. Generalized isolation level definitions. In *Proceedings of 16th International Conference on Data Engineering (Cat. No.00CB37073)*, pages 67–78, 2000.

[ALR+17] Mihnea Andrei, Christian Lemke, Günter Radestock, Robert Schulze, Carsten Thiel, Rolando Blanco, Akanksha Meghlan, Muhammad Sharique, Sebastian Seifert, Surendra Vishnoi, et al. Sap hana adoption of non-volatile memory. *Proceedings of the VLDB Endowment*, 10(12):1754–1765, 2017.

[AMDM07] Daniel J Abadi, Daniel S Myers, David J DeWitt, and Samuel R Madden. Materialization strategies in a column-oriented dbms. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 466–475. IEEE, 2007.

[AMF06] Daniel Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006*

ACM SIGMOD international conference on Management of data, pages 671–682. ACM, June 2006.

[AMH08] Daniel J. Adadi, Samuel R. Madden, and Nabil Hachem. Column-stores vs. row-stores: how different are they really? In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 967–980. ACM, June 2008.

[Apaa] ApacheSoftwareFoundation. Documentation Apache Parquet. `https://parquet.apache.org/documentation/latest/`.

[Apab] ApacheSoftwareFoundation. LanguageManual ORC. `https://cwiki.apache.org/confluence/display/Hive/LanguageManual+ORC`.

[APM16] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. Bridging the archipelago between row-stores and column-stores for hybrid workloads. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 583–598, New York, NY, USA, 2016. Association for Computing Machinery.

[AS10] Hiroyuki Akinaga and Hisashi Shima. Resistive random access memory (reram) based on metal oxides. *Proceedings of the IEEE*, 98(12):2237–2251, 2010.

[Bay98] Roberto J. Bayardo. Efficiently mining long patterns from databases. *SIGMOD Rec.*, 27(2):85 – 93, jun 1998.

[BBD+15] Ute Baumbach, Patric Becker, Uwe Denneler, Eberhard Hechler, Wolfgang Hengstler, Steffen Knoll, Frank Neumann, Guenter Georg Schoellmann, Khadija Souissi, and Timm Zimmermann. Accelerating Data Transformation with IBM DB2 Analytics Accelerator for z/OS. `http://www.redbooks.ibm.com/redbooks/pdfs/sg248314.pdf`, December 2015. Retrieved April 29, 2016.

[BBG+07] Hal Berenson, Philip Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. *CoRR*, abs/cs/0701157, 01 2007.

[BGMS92] Yuri Breitbart, Hector Garcia-Molina, and Avi Silberschatz. Overview of multidatabase transaction management. volume 1, pages 181–240, Berlin, Heidelberg, October 1992. Springer-Verlag.

[BHEF11] M. A. Bornea, O. Hodson, S. Elnikety, and A. Fekete. One-copy serializability with snapshot isolation under the hood. In *2011 IEEE 27th International Conference on Data Engineering*, pages 625–636, 2011.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[Cah27] Michael James Cahill. *Serializable Isolation for Snapshot Databases*. PhD thesis, 2009-08-27.

[CFG+11] Richard Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi Kuno, Raghunath Nambiar, Thomas Neumann, Meikel Poess, Kai-Uwe Sattler, Michael Seibold, Eric Simon, and Florian Waas. The mixed workload ch-benchmark. In *Proceedings of the Fourth International Workshop on Testing Database Systems*, DBTest '11, New York, NY, USA, 2011. Association for Computing Machinery.

[Coua] Transaction Processing Performance Council. Tpc benchmark c. `https://www.tpc.org/tpcc/`.

[Coub] Transaction Processing Performance Council. Tpc benchmark h. `https://www.tpc.org/tpch/`.

[CRF09] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. Serializable isolation for snapshot databases. *ACM Trans. Database Syst.*, 34(4), dec 2009.

[CSP+12] Youngdon Choi, Ickhyun Song, Mu-Hui Park, Hoeju Chung, Sanghoan Chang, Beakhyoung Cho, Jinyoung Kim, Younghoon Oh, Duckmin Kwon, Jung Sunwoo, et al. A 20nm 1.8 v 8gb pram with 40mb/s program bandwidth. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2012 IEEE International*, pages 46–48. IEEE, 2012.

[DFI+13] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: Sql server' s memory-optimized oltp engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD ' 13, pages 1243–1254, New York, NY, USA, 2013. Association for Computing Machinery.

[doc22] Azure Cosmos DB documentation. Azure synapse link for azure cosmos db: Near real-time analytics use cases. `https://docs.microsoft.com/en-us/azure/cosmos-db/synapse-link-use-cases`, Febrary 2022. Accessed: 2022-04-08.

[DPCCM13] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proc. VLDB Endow.*, 7(4):277–288, December 2013.

[EPZ05] S. Elnikety, F. Pedone, and W. Zwaenepoel. Database replication using generalized snapshot isolation. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*, pages 73–84, 2005.

[FLO+05] Alan Fekete, Dimitrios Liarokapis, Elizabeth O' Neil, Patrick O' Neil, and Dennis Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, June 2005.

[FOO04] Alan Fekete, Elizabeth O'Neil, and Patrick O'Neil. A read-only transaction anomaly under snapshot isolation. *SIGMOD Rec.*, 33(3):12–14, September 2004.

[HLH+11] Y. He, R. Lee, Y. Huai, Z. Shao, N. Jain, X. Zhang, and Z. Xu. Rcfile: A fast and space-efficient data placement structure in mapreduce-based warehouse systems. In *2011 IEEE 27th International Conference on Data Engineering*, pages 1199–1208, April 2011.

[HZN+10] Sándor Héman, Marcin Zukowski, Niels J Nes, Lefteris Sidirourgos, and Peter Boncz. Positional update handling in column stores. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 543–554. ACM, 2010.

[JHFR11] Hyungsoo Jung, Hyuck Han, Alan Fekete, and Uwe Röhm. Serializable snapshot isolation for replicated databases in high-update scenarios. *Proc. VLDB Endow.*, 4(11):783–794, August 2011.

[KN11] A. Kemper and T. Neumann. Hyper: A hybrid oltp olap main memory database system based on virtual memory snapshots. In *2011 IEEE 27th International Conference on Data Engineering*, pages 195–206, 2011.

[KPB+19] Andreas Kipf, Varun Pandey, Jan Böttcher, Lucas Braun, Thomas Neumann, and Alfons Kemper. Scalable analytics on fast data. *ACM Trans. Database Syst.*, 44(1), jan 2019.

[LBD+11] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. High-performance concurrency control mechanisms for main-memory databases. *Proc. VLDB Endow.*, 5(4):298–309, December 2011.

[LFV+12] 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.

[LMM+13] Juchang Lee, Michael Muehle, Norman May, Franz Faerber, Vishal Sikka, Hasso Plattner, Jens Krüger, and Martin Grund. High-performance transaction processing in SAP HANA. *IEEE Data Eng. Bull.*, 36(2):28–33, 2013.

[MGBA17] 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*, SIGMOD '17, pages 37–50, New York, NY, USA, 2017. Association for Computing Machinery.

[MKN11] Henrik Mühe, Alfons Kemper, and Thomas Neumann. How to efficiently snapshot transactional data: Hardware or software controlled? In *Proceedings of the Seventh International Workshop on Data Management on New Hardware*, DaMoN '11, pages 17–26, New York, NY, USA, 2011. Association for Computing Machinery.

[NMK15] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. Fast serializable multi-version concurrency control for main-memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD ' 15, pages 677–689, New York, NY, USA, 2015. Association for Computing Machinery.

[OTT17] 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, SIGMOD '17, pages 1771–1775, New York, NY, USA, 2017. Association for Computing Machinery.

[Pap86] Christos Papadimitriou. *Theory of Database Concurrency Control*. Computer science Press., 1986.

[Pap15] Oracle White Paper. Oracle goldengate 12 c : Real-time access to real-time information. March 2015.

[PAZN05] Boncz Peter, A., Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proceedings of Conference of Innovative Database Research*, pages 225–237. CIDR, 2005.

[PG12] Dan R. K. Ports and Kevin Grittner. Serializable snapshot isolation in postgresql. *Proc. VLDB Endow.*, 5(12):1850–1861, August 2012.

[Pla09] 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*, SIGMOD '09, pages 1–2, New York, NY, USA, 2009. Association for Computing Machinery.

[PMJA01] Sriram Padmanabhan, Timothy Malkemus, Anant Jhingran, and Ramesh Agarwal. Block oriented processing of relational database operations in modern computer architectures. In *Data Engineering, 2001. Proceedings. 17th International Conference on*, pages 567–574. IEEE, April 2001.

[RBZ13] Bogdan Răducanu, Peter Boncz, and Marcin Zukowski. Micro adaptivity in vectorwise. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1231–1242. ACM, June 2013.

[RCAA20] Aunn Raza, Periklis Chrysogelos, Angelos Christos Anadiotis, and Anastasia Ailamaki. Adaptive htap through elastic resource scheduling. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, pages 2043–2054, New York, NY, USA, 2020. Association for Computing Machinery.

[SAB+05] Mike Stonebraker, Daniel J. Asadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-Store: A

Column-Oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases*, pages 553–564. VLDB Endowment, August 2005.

[Sc18] Michael Stonebraker and Uundefinedur Çetintemel. *"One Size Fits All": An Idea Whose Time Has Come and Gone*, pages 441–462. Association for Computing Machinery and Morgan & Claypool, 2018.

[ŚE09] Dominik Ślezak and Victoria Eastwood. Data warehouse technology by infobright. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 841–846. ACM, 2009.

[SHY17] Takamitsu Shioi, Kenji Hatano, and Haruo Yokota. Scmat: A mechanism presuming scms to efficiently enable both olap and oltp. In *2017 IEEE International Congress on Big Data (BigData Congress)*, pages 313–320, 2017.

[SKA+24] Takamitsu Shioi, Takashi Kambayashi, Suguru Arakawa, Ryoji Kurosawa, Satoshi Hikida, and Haruo Yokota. Read-safe snapshots: An abort/wait-free serializable read method for read-only transactions on mixed oltp/olap workloads. *Information Systems*, 124:102385, 2024.

[SPP16] Alex Skidanov, Anders J Papito, and Adam Prout. A column store engine for real-time streaming analytics. In *Data Engineering (ICDE), 2016 IEEE 32nd International Conference on*, pages 1287–1297. IEEE, May 2016.

[SW00] Ralf Schenkel and Gerhard Weikum. Integrating snapshot isolation into transactional federation. In *Proceedings of the 7th International Conference on Cooperative Information Systems*, CooplS '02, pages 90–101, Berlin, Heidelberg, 2000. Springer-Verlag.

[SWWW00] Ralf Schenkel, Gerhard Weikum, Norbert Weißenberg, and Xuequn Wu. Federated transaction management with snapshot isolation. In Gunter Saake, Kerstin Schwarz, and Can Türker, editors, *Transactions and Database Dynamics*, pages 1–25, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[TSM+20] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. Cockroachdb: The resilient geo-distributed sql database. In

Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, SIGMOD '20, pages 1493–1509, New York, NY, USA, 2020. Association for Computing Machinery.

[TZK+13] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 18–32, New York, NY, USA, 2013. Association for Computing Machinery.

[UGA+09] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann. Predictable performance for unpredictable workloads. *Proc. VLDB Endow.*, 2(1):706–717, August 2009.

[WOS02] Kesheng Wu, Ekow J Otoo, and Arie Shoshani. Compressing bitmap indexes for faster search operations. In *Scientific and Statistical Database Management, 2002. Proceedings. 14th International Conference on*, pages 99–108. IEEE, July 2002.

[WV01] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[ZNB08] Marcin Zukowski, Niels Nes, and Peter Boncz. DSM vs. NSM: CPU performance tradeoffs in block-oriented query processing. In *Proceedings of the 4th international workshop on Data management on new hardware*, pages 47–54. ACM, June 2008.

Publications

Journals (refereed)

- Takamitsu Shioi, Takashi Kambayashi, Suguru Arakawa, Ryoji Kurosawa, Satoshi Hikida, and Haruo Yokota. "Read-safe snapshots: An abort/wait-free serializable read method for read-only transactions on mixed oltp/olap workloads", *Information Systems*, 124:102385, 2024.9.

Proceedings on international conference (refereed)

- Takamitsu Shioi, Kenji Hatano, Haruo Yokota. "SCMAT: A Mechanism Presuming SCMs To Efficiently Enable both OLAP And OLTP", *Proceeding of the 6th IEEE International Congress on Big Data (BigData Congress 2017)*, pp. 313–320, 2017.6.