Sequoia RAIDb-3: a new model for data distribution and replication using commodity systems

Rostislav Axamitnyy

A Thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fullfilment of the Requirements
for the Degree of Master of Science at
Concordia University
Montreal, Quebec, Canada

April, 2019

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: Rostislav Axamitnyy

Entitled: Sequoia RAIDb-3: a new model for data distribution and replication using commodity systems

and submitted in partial fulfillment of the requirements for the degree of

Master of Science

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

signed by the examining committee:

_____ Chair
Dr. G. Butler

_____ Examiner
Dr. J. Rilling

_____ Examiner
Dr. N. Tsantalis

_____ Supervisor
Dr. T. Eavis

Approved By _____
Chair of Department or Graduate Program Director

_____20___ _____
Dean of Faculty

ABSTRACT

Sequoia RAIDb-3: a new model for data distribution and replication using commodity systems

Rostislav Axamitnyy

Unprecedented growth in the amount of data generated and used in the modern world has made distributed databases an important target for contemporary research. Today, advances in distributed databases embrace a wide range of concepts and ideas including, but not limited to, fragmentation, replication, distributed transactions and distributed concurrency control.

One novel idea in this area was the introduction of the Redundant Array of Inexpensive Databases (RAIDb) initially proposed by the authors of Sequoia, a Java-based clustering middle-ware framework. On a conceptual level, RAIDb is similar to RAID arrays of disks; however, in contrast to traditional RAID, RAIDb utilizes an array of individual databases. The objective of RAIDb is to provide improved performance and fault tolerance relative to a single database while preserving the abstraction of a standard SQL DBMS.

This thesis extends the functionality of Sequoia and proposes a distribution model based upon full horizontal fragmentation. We refer to this new design as RAIDb-3. We discuss details of the implementation and support its validity with an extensive suite of test cases.

ACKNOWLEDGEMENTS

I wish to express my gratitude to my supervisor, Dr. Todd Eavis. This work will not be possible without his priceless support, encouragement and inspiration.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Distributed databases

Today, distributed databases play an ever increasing role in information technology and form the basis for many large data management systems. This trend will definitively continue due to the enormous growth of stored and processed data, coupled with the demand for data management system scalability, high availability and fault tolerance. Distributed databases address these challenges through replication and fragmentation - two concepts forming the foundation of its data model. In short, replication ensures a certain level of data redundancy within the database, thereby eliminating the single point of failure. Fragmentation, or partitioning, is the process of splitting data among nodes of the distributed database. It provides for increased performance and high availability of distributed data.

Despite its obvious benefits, the concept of a distributed database is associated with a certain degree of inherent complexity. In contrast to conventional database management systems, distributed databases need to address such challenges as establishing concurrency control, handling distributed queries and transactions, maintaining data integrity, etc. All these issues, as well as fragmentation and replication,

have been important topics for research. Many theoretical approaches, along with concrete implementations, have been proposed in recent years. Some of these new solutions devise completely new models which are not directly related to any existing technology or product. This group is represented by such novel technologies as NoSQL data engines, XML databases and NewSQL databases. Deployment of these solutions typically implies complete redesign and replacement of existing data management environments. However, there is also another direction for research that focuses on integrating the core principles of distributed databases into existing data management infrastructures. This can be achieved by abstracting away the complexity of a clustered database to produce the appearance of a conventional relational database.

One such system is the Sequoia RAIDb clustering middleware. Sequoia builds upon the concept of a Redundant Array of Inexpensive Databases (RAIDb). This approach addresses a number of the challenges of storing and processing distributed data, such as fault-tolerance and high performance, while at the same time providing for easy integration into existing SQL-based data analytics solutions. In fact, Sequoia offers full distribution transparency, providing applications that utilize Sequoia with a complete abstraction of conventional SQL database management systems. Moreover, Sequoia is an open-source project and, as such, is fully extensible at the source code level.

In this thesis we examine the existing Sequoia's RAIDb solution. More importantly, we extend its current functionality in order to achieve more effective data distribution and parallelism among independent database nodes.

1.2 RAIDb-3 - full horizontal fragmentation in Sequoia

Sequoia introduces the concept of a Redundant Array of Inexpensive Databases (RAIDb), which closely resembles the existing concept of RAID (Redundant Array of Inexpensive Disks) that achieves scalability and high availability of disk subsystems at a low cost. While RAID aggregates multiple inexpensive disk drives into an array of disks to obtain performance, capacity and reliability properties that exceed that of a single large drive [21] - RAIDb combines multiple database instances into an array of databases aiming for a low cost system providing better performance and fault tolerance than a single database [7]. The primary target for RAIDb deployment is low-cost commodity hardware and software such as clusters of workstations using open-source databases.

The RAIDb model is implemented in JDBC-compatible, Java mddleware called the Sequoia RAIDb controller. The RAIDb controller hides all details of the RAIDb implementation and provides applications with a pure abstraction of conventional SQL database management systems (DBMS). This abstraction permits easy deployment of Sequoia in existing SQL-based environments. At the same time, the RAIDb controller utilizes conventional JDBC-compatible SQL DBMSes as building blocks of the RAIDb array. We will use the terms *database back-end* and *node* to denote these individual databases. Ultimately, this means that a RAIDb array can be constructed from widely available open-source databases such as HSQLDB. Furthermore, the standard JDBC API allows nodes to be spread across the network.

In fact, the authors of the original Sequoia implementation proposed three distinct RAIDb levels.

- **RAIDb-0:** provides for partitioning of database tables among database back-ends. Partitioning applies to complete tables only, meaning that a table itself may not be partitioned across nodes but different tables can be distributed on distinct nodes.

- **RAIDb-1:** offers full replication, meaning that every database back-end holds the complete database.

- **RAIDb-2:** provides partial replication of tables combined with partitioning.

It is important to note, however, that the data distribution model implemented in Sequoia only supports table level partitioning. This fact significantly undermines one of the fundamental goals of data partitioning, namely performance, as transaction processing strongly depends on both the properties of the transaction and the layout of the table data amongst nodes. In fact, in certain scenarios no table partitioning might occur at all and, as a result, only a single node might be involved in execution of the transaction. We address this challenge in our work, as we aim to devise a more sophisticated model for data distribution within the Sequoia RAIDb framework.

In this work we propose a model for full horizontal partitioning in the RAIDb controller which we call RAIDb-3. Our model allows for partitioning a table along its rows so that these rows are evenly spread across all database back-ends. In order to achieve this level of partitioning we introduce a new layer of data management into Sequoia, which we refer to as the *Key Manager*. Furthermore, almost every core component of the Sequoia DBMS is modified in order to support the requirements of the new model. Below, we list the most critical functionality affected by the implementation of RAIDb-3.

- *Virtual database.* We transform the structure of the virtual database representation in Sequoia for the purpose of delivering a more fine-grained view of the virtual meta-data.

- *SQL parsing.* An extended form of syntax parsing has been implemented so as to provide proper handling of table constraints.

- *Query processing.* We implement distribution-aware logic for processing core SQL queries, such as `CREATE TABLE`, `INSERT` and `SELECT`.

- *Load balancing.* We introduce a *round-robin* mechanism so as to allow even distribution of records among nodes. In addition, we modify the form of node interaction to permit query execution concurrency among nodes.

- *Result handling.* We extend result handling structures so as to provide for the processing of partial results and for communication with the Key Manager.

- *JDBC driver.* We adapt Sequoia's JDBC driver interface to permit interaction with our internal result handling structures.

Moreover, we introduce several additional architectural components that are necessary for operation of the RAIDb-3 model. These elements include:

- *Key Manager.* A component that manages the processing of table constraints and generates internal service queries.

- *RaControllerResultSet.* This is an internal structure developed to store and aggregate partial results received concurrently from back-end nodes.

1.3 Experimental results

Experimental results have shown that our RAIDb-3 model represents a viable option for storing and manipulating large data populations using conventional hardware and software platforms as database back-ends. We also demonstrate that for certain types of workloads, RAIDb-3 exhibits better performance than standalone databases. Perhaps more importantly, we note that the performance of the RAIDb-3 controller increases relative to standalone databases with growth in the data set. That said, we also discovered that due to implementation constraints, performance of RAIDb-3 remains below that of standalone databases for certain queries. In these cases, we suggest possible options for future work.

1.4 Thesis structure

The thesis is organized as follows. Chapter 2 provides an overview of some of the most important concepts in the field of distributed databases. It also briefly outlines notable research in this area. Chapter 3 presents an overview of Sequoia's RAIDb clustering middle-ware. It then proposes a new approach for implementing full horizontal fragmentation, a model we call RAIDb-3. Chapter 4 presents experimental results from various perspectives. Chapter 5 offers final conclusions and suggests directions for future work.

Chapter 2

Background Material

2.1 Introduction

The trend toward distributing data and computation began decades ago, but has received significant attention in recent years. The principles of distributing computation are almost omnipresent in the modern applications of computer science, ranging from the distribution of complex mathematical computations on multi-core GPUs to the evolution of cryptocurrencies that simply do not exist in the non-distributed context.

This same trend also changed the data storage and data management landscape previously dominated by stand-alone, high-end database management systems. Distributed databases became an answer to two challenges – the ever-increasing amount of data and the increasing degree of distribution of business processes. Moreover, the latter also includes geographical distributions when several inter-connected business units are spread across various physically distinct areas, as well as technological distribution whereby various services reside on heterogeneous computational units

connected through a network (e.g. web-hosting, computational clouds and containerized processes). Besides addressing technological and organizational challenges, distributed databases are meant to deal with such longstanding issues as reliability, high-availability and database system responsiveness.

2.2 Relational databases and SQL

In this work we primarily elaborate on the topic of distributed databases. Most existing databases today are based on the relational model [22], which was first described by Codd in [6]. In the relational context, all data is organized in relations, or tables. In turn, relations are composed of attributes, or columns, and tuples, also referred to as rows. Hence, a relation represents a distinct entity (e.g., invoice) characterized by attributes (e.g., date and price). Instances of a given entity are stored in the relation's tuples (e.g., invoices for books).

The relational model allows rows of distinct tables to be related to each other. In order to facilitate this process each table can have a special attribute called the *primary key* that allows each row to be assigned a unique value (e.g., it might be order identification number in the case of the invoice example). A *primary key* from one table can correspond to a matching *foreign key* in a second table.

The relational model is directly associated with relational database management systems (RDBMS) which are responsible for the implementation of the concepts of the relation model, as well as for establishing the efficient and accurate operation of the database system. The primary means of interacting with RDBMSes is the Structured Query Language (SQL), the de facto standard specially designed for querying and maintaining RDBMSes. SQL defines a variety of syntactical constructs to express or

support the concepts of the relational model and to mimic the operators defined in the *relational algebra*, the formal mathematical foundation at the heart of the relational model.

Below, we provide a very brief introduction to the relational algebra as it pertains to data handling in RDBMSes. A more extensive overview of the SQL related elements is provided in Chapter 3.

Relational algebra is associated with the operators defined in basic set theory, such as set union, set difference, and Cartesian product. It also adds a few additional operators which we list below.

- *Selection:* A selection is a unary operation that identifies tuples from a relation whose attributes meet the selection criteria. The selection criteria is usually expressed as a predicate. The selection operator is written as $\sigma_{\varphi}(R)$, where R is the relation and φ is a propositional formula. A propositional formula is composed of predicates applied on attributes of R, and may be concatenated by logical operators, such as `AND`, `OR` or `NOT`.

  As an example, we may consider the relation R depicted in Table 2.1.

  Moreover, we can define a propositional formula in the following way

  $$\varphi = AGE > 30 \wedge PROVINCE = QC$$

  This selection operator would then produce the result depicted in Table 2.2

- *Projection:* A projection is a unary operation that extracts a subset of the columns in a relation. The projection operator is written as $\Pi_{a_1,\ldots,a_n}(R)$ where $a_1, \ldots, a_n$ is a set of attribute names.

Table 2.1: Relation `CUSTOMER`

NAME	AGE	SEX	PROVINCE	LOCATION
Jen Nemhauser	38	M	QC	Dorval
Sydel Jephthah	56	M	MN	Dauphin
Chrystal Norvall	30	F	NU	Iqaluit
Binni Noonberg	35	M	NL	Corner Brook
Garland Eatton	38	M	NS	Halifax
Berrie Fougere	36	F	QC	Blainville
Wenda Brubaker	71	F	ON	Stratford
Nan Cornell	52	M	BC	Kimberley

Table 2.2: Result of selection operator applied to relation `CUSTOMER`

NAME	AGE	SEX	PROVINCE	LOCATION
Jen Nemhauser	38	M	QC	Dorval
Berrie Fougere	36	F	QC	Blainville

As an example, we may again consider relation R presented in Table 2.1. If we define a set of attributes as:

$$\{a_1, \ldots, a_n\} = \{AGE, SEX\}$$

The projection operator would then produce the result depicted in Table 2.3

- *Joins:* The relational algebra defines a number of join operators. The most generic type of join operator is a *natural join* which we will describe here (other join operators extend the basic concept of natural join in various ways). A natural join is a binary operator that combines attributes of two relations into one. The natural join is written as $R \bowtie S$ where R and S are relations.

  As an example we will again consider relation R presented in Table 2.1. Furthermore, we define a second relation S, presented in Table 2.4.

Table 2.3: Result of projection operator applied to relation `CUSTOMER`

AGE	SEX
38	M
56	M
30	F
35	M
38	M
36	F
71	F
52	M

Table 2.4: Relation `INVOICE`

NAME	TOTAL	DATE
Chrystal Norvall	91.89	2016-02-04
Jen Nemhause	24.81	2017-07-20
Sydel Jephthah	122.63	2008-07-19
Binni Noonberg	128.8	2016-02-07
Garland Eatton	56.25	2015-07-20
Wenda Brubaker	100.00	2014-08-18
Jen Nemhause	25.50	2017-10-13
Nan Cornell	63.40	2017-02-03
Nan Cornell	89.66	2016-01-06
Berrie Fougere	68.70	2016-05-22
Sydel Jephthah	15.30	2010-12-07
Binni Noonberg	93.00	2012-10-23
Sydel Jephthah	80.00	2013-11-25

A natural join operation on R and S would then produce the result depicted in Table 2.5

Table 2.5: Result of natural join operator applied to relations `CUSTOMER` and `INVOICE`

AGE	SEX	PROVINCE	LOCATION	NAME	TOTAL	DATE
30	F	NU	Iqaluit	Chrystal Norvall	91.89	2016-02-04
38	M	QC	Dorval	Jen Nemhause	24.81	2017-07-20
56	M	MN	Dauphin	Sydel Jephthah	122.63	2008-07-19
35	M	NL	Corner Brook	Binni Noonberg	128.8	2016-02-07
38	M	NS	Halifax	Garland Eatton	56.25	2015-07-20
71	F	ON	Stratford	Wenda Brubaker	100.00	2014-08-18
38	M	QC	Dorval	Jen Nemhause	25.50	2017-10-13
52	M	BC	Kimberley	Nan Cornell	63.40	2017-02-03
52	M	BC	Kimberley	Nan Cornell	89.66	2016-01-06
36	F	QC	Blainville	Berrie Fougere	68.70	2016-05-22
56	M	MN	Dauphin	Sydel Jephthah	15.30	2010-12-07
35	M	NL	Corner Brook	Binni Noonberg	93.00	2012-10-23
56	M	MN	Dauphin	Sydel Jephthah	80.00	2013-11-25

2.3 Distributed database design overview

Özsu et al [33] define a distributed database as follows: "...a collection of multiple, logically interrelated databases distributed over a computer network. A distributed database management system (distributed DBMS) is then defined as the software system that permits the management of the distributed database and makes the distribution transparent to the users."

2.3.1 Architecture

From an architectural perspective, the design of a distributed DBMS may be characterized as the exploitation and integration of the autonomy of local systems, their distribution, and their heterogeneity.

- *Autonomy:* implies distribution of control over data or, alternatively, reflects the degree of independence of the DBMS. Autonomy may vary from total isolation, when each DBMS comprising a distributed system is deprived of any information regarding other participating DBMSes or the topology of the distributed system, to cases in which DBMSes are logically integrated.

- *Distribution:* determines the level of distribution of data. Two notable data distribution classes are (1) client/server distribution, as shown in Figure 2.1, where data is stored within server nodes and accessed by clients; (2) peer-to-peer distribution, in which there is no functional distinction among nodes and each peer mimics the functionality of the full DBMS while interacting with other peers.

- *Heterogeneity:* heterogeneity may assume various forms such as variances in hardware platforms, operating systems, communication protocols, data representation formats, etc.

2.3.2 Design objectives

Distributed database systems are designed to comply with the following objectives:

- Transparency of distribution and replication;

- Reliability of execution of distributed transactions;

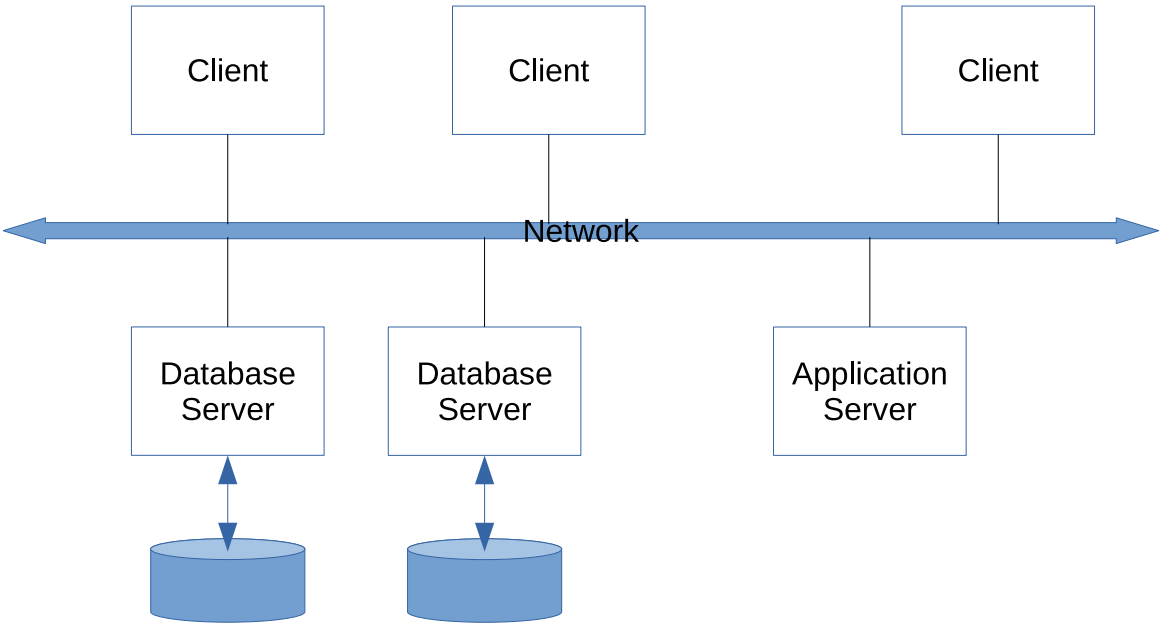

Figure 2.1: Distributed client/server system.

Below, we elaborate in more detail on these two themes.

Transparency of distribution and replication implies a level of abstraction in which all details of the implementation are hidden from the user of the distributed DBMS. The following forms of transparency are included [33]:

- *Data independence:* this form of transparency assures that neither modifications to the logical nature (i.e. the schema) nor the psychical nature (i.e. the structure of physical data storage) should be visible to the user of the distributed system.

- *Network transparency:* encapsulates the topology of the network organization of distributed system so that neither the existing specificity of network structure nor its possible modifications affect the user of the distributed DBMS.

- *Replication transparency:* this form of transparency assures that the methodology of replication, as well as the management of replicas, remains hidden from

the user. We will cover the topic of replication later in this chapter.

- *Fragmentation transparency:* provides the user of the distributed database with an abstraction of an integrated data view, while hiding possible data fragmentation. The topic of fragmentation will also be examined in detail later.

The reliability of execution of distributed transactions ensures the atomicity of transactions on distributed databases. Specifically, while executing transactions on a distributed DBMS, the original transaction might be decomposed into several sub-transactions which are then executed on various nodes of the distributed system, with the results combined at a later point. However, any of these sub-transactions may fail; therefore, reliability of execution guarantees that either the complete transaction is executed successfully, including all of its sub-transactions, or if any single step of the transaction should fail, the database remains intact so that consistency is preserved.

2.3.3 Data placement and fragmentation

One of the crucial questions in the design of distributed databases is how data is placed across nodes. At present, two main approaches to placement of data are distinguished:

- Replication
- Fragmentation or partitioning

Replication is a process of creating "local" copies of data that otherwise are stored remotely – or more generally a process of storing full or partial copies of data on all or some distributed nodes. The motivation behind applying replication includes:

- *Increased data availability:* replication eliminates the existence of a single point of failure.

- *Increased performance:* replication can minimize response time by identifying the optimal path to the requested data and/or through increased parallelism among replicated nodes.

- *Scalability/Flexibility:* replication supports system growth (or reduction) by modifying the number of replicas.

At the same time, replication poses considerable challenges for the design of data management systems in terms of data updates among replicas. Presently, there are two approaches to address the issue of update propagation: eager and lazy replication [30]. In the case of eager replication, all replicas are updated within the scope of the distributed update transaction. Here, the execution of the update is synchronous. For lazy replication, the update transaction is executed asynchronously among replicas (i.e., the transaction is committed on a local node and then the process of notifying all other replicas is initiated). Both approaches have their pros and cons. Eager replication, for example, can handle conflicts naturally by aborting the global transaction; however, responsiveness might deteriorate considerably as a result of network issues or node hardware/software failure. Lazy replication, on the other hand, does not provide explicit conflict resolution mechanisms and requires conflict resolution to be perfromed by resynchronization [14].

Fragmentation is another fundamental process underlying the concept of distributed databases. During this process, the original relation stored in the database is divided into a number of sub-relations, or fragments, which can then be stored on

different nodes. Traditionally, the motivation for fragmentation can be summarized by the following points:

- *Performance:* due to the fact that data is spread among various fragments or partitions, it is possible to execute transactions in a concurrent, parallel fashion.
- *Alleviate negative effects of replication:* replication considerably increases storage utilization because of its inherent redundancy; however, fragmentation allows for a reduction in the footprint, as each replica will store only a fragment of the original relation, rather than the full relation;
- *Address needs of business processes:* usually DBMS applications make use of only a subset of a relation. Hence the full relation is not necessarily the most appropriate unit of distribution. Instead, subsets of the relation tend to be a more natural unit of distribution.

In the world of relational databases, a relation is in essence a table. Therefore, two alternatives are available to divide a table into sub-tables: horizontal and vertical. These two approaches form basic fragmentation strategies. Furthermore, there is an option to nest fragments, thereby providing for hybrid fragmentation. In fact, there are number of different approaches and algorithms used to accomplish fragmentation – we will provide an overview of existing methods later in this chapter.

In terms of horizontal fragmentation, we note that it is performed on the tuples of the original relation so that the resulting fragments contain subsets of the tuples of the original relation. Fragmentation is performed by applying a specific predicate P_i on the original relation. In the case where a predicate is defined on the same relation on which it is applied, horizontal fragmentation is characterized as *primary*

horizontal fragmentation, and fragments are derived according to the formula:

$$R_i = P_i(R), 1 \leq i \leq \omega$$

where:

R – original relation

R_i – subset of original relational

P_i – predicate

ω – degree of fragmentation

For example, we may consider the original relation `CUSTOMER`, as depicted in Table 2.6

Table 2.6: Relation `CUSTOMER`

NAME	ID	AGE	SEX	PROVINCE	LOCATION
Jen Nemhauser	0	38	M	QC	Dorval
Sydel Jephthah	1	56	M	MN	Dauphin
Chrystal Norvall	2	30	F	NU	Iqaluit
Binni Noonberg	3	35	M	NL	Corner Brook
Garland Eatton	4	38	M	NS	Halifax
Berrie Fougere	5	36	F	QC	Blainville
Wenda Brubaker	6	71	F	ON	Stratford
Nan Cornell	7	52	M	BC	Kimberley

In the relation `CUSTOMER`, the attribute `ID` is the primary key. For this relation we might define fragments based on age (i.e., the attribute `AGE`). Furthermore, we could define predicates as follows:

$\texttt{CUSTOMER}_1$: $P_1 = AGE < 38$

$\texttt{CUSTOMER}_2$: $P_2 = 38 \leq AGE < 49$

$\texttt{CUSTOMER}_3$: $P_3 = AGE \geq 49$

This configuration results in three fragments – $\texttt{CUSTOMER}_1$, $\texttt{CUSTOMER}_2$, $\texttt{CUSTOMER}_3$.

Table 2.7: Fragment $\texttt{CUSTOMER}_1$

NAME	ID	AGE	SEX	PROVINCE	LOCATION
Chrystal Norvall	2	30	F	NU	Iqaluit
Binni Noonberg	3	35	M	NL	Corner Brook
Berrie Fougere	5	36	F	QC	Blainville

Table 2.8: Fragment $\texttt{CUSTOMER}_2$

NAME	ID	AGE	SEX	PROVINCE	LOCATION
Jen Nemhauser	0	38	M	QC	Dorval
Garland Eatton	4	38	M	NS	Halifax

Table 2.9: Fragment $\texttt{CUSTOMER}_3$

NAME	ID	AGE	SEX	PROVINCE	LOCATION
Sydel Jephthah	1	56	M	MN	Dauphin
Wenda Brubaker	6	71	F	ON	Stratford
Nan Cornell	7	52	M	BC	Kimberley

Additionally, a predicate can be associated with a relation without the predicate

being directly applied to the original relation itself – this type of horizontal fragmentation is classified as *derived* horizontal fragmentation. Hence, the resulting fragment is defined through a semi-join:

$$R_i = R \ltimes S_i, 1 \leq i \leq \omega$$

where:

R – original relation

S_i – a subset of another relation, obtained by applying the predicate P_i to the relation S , i.e. $S_i = P_i(S)$

R_i – resulting fragment of relation R

For example, we can consider the example listed above and further extend it with the additional relation `DISCOUNT`, depicted in Table 2.10.

Table 2.10: Fragment `DISCOUNT`

PROVINCE	DISCOUNT_P
QC	15
MN	10
NU	8
NL	5
NS	15
ON	5
BC	0

For relation `DISCOUNT`, we may define a predicate on the attribute `DISCOUNT_P` as follows:

$\texttt{DISCOUNT}_1$: $P_1 = DISCOUNT\_P < 5$

$\texttt{DISCOUNT}_2$: $P_2 = 5 \leq DISCOUNT\_P < 10$

$\texttt{DISCOUNT}_3$: $P_3 = DISCOUNT\_P \geq 10$

Applying these predicates on the relation DISCOUNT produces three sub-relations: $\texttt{DISCOUNT}_1$, $\texttt{DISCOUNT}_2$ and $\texttt{DISCOUNT}_3$. The resulting fragments of relation CUSTOMER are consequently defined as follows:

$CUSTOMER_1 = CUSTOMER \ltimes DISCOUNT_1$

$CUSTOMER_2 = CUSTOMER \ltimes DISCOUNT_2$

$CUSTOMER_3 = CUSTOMER \ltimes DISCOUNT_3$

The results of the derived horizontal fragmentation are depicted in tables 2.11 through 2.13.

Table 2.11: Fragment $\texttt{CUSTOMER}_1$

NAME	ID	AGE	SEX	PROVINCE	LOCATION
Nan Cornell	7	52	M	BC	Kimberley

Table 2.12: Fragment $\texttt{CUSTOMER}_2$

NAME	ID	AGE	SEX	PROVINCE	LOCATION
Chrystal Norvall	2	30	F	NU	Iqaluit
Binni Noonberg	3	35	M	NL	Corner Brook
Wenda Brubaker	6	71	F	ON	Stratford

Table 2.13: Fragment CUSTOMER$_3$

NAME	ID	AGE	SEX	PROVINCE	LOCATION
Jen Nemhauser	0	38	M	QC	Dorval
Sydel Jephthah	1	56	M	MN	Dauphin
Garland Eatton	4	38	M	NS	Halifax
Berrie Fougere	5	36	F	QC	Blainville

Selection of predicates for fragmentation is complex and depends heavily on the context in which the distributed database operates (e.g., devising a fragmentation approach in which certain join characteristics are observed or in which a higher level of parralelization can be achieved). At present, the issue of fragmentation is a field of extensive research and several algorithms have been proposed to produce optimal fragmentation - some of the most notable algorithms can be found in [33].

As noted, a second approach is *vertical* fragmentation. In contrast to horizontal fragmentation, vertical fragmentation is performed along attributes. The result of vertical fragmentation of a relation R is the fragment set R_1, R_2, ..., R_n, each of which contains a subset of attributes of the original relation R, as well as the primary key of R. If we consider the previously defined relation CUSTOMER, depicted in Table 2.6, one possible vertical fragmentation into two sub-relations CUSTOMER$_1$ and CUSTOMER$_2$ might assume the form depicted in 2.14 and 2.15. Here, the Attribute ID in relation CUSTOMER is the primary key.

When devising fragmentation techniques, execution time of the user application that runs on the fragments is considered an optimization criteria. Therefore, the objective of vertical fragmentation is to define a scheme that minimizes execution time. Often, it is desirable that many application will run on only one fragment.

Table 2.14: Fragment $\texttt{CUSTOMER}_1$

NAME	ID	AGE	SEX
Jen Nemhauser	0	38	M
Sydel Jephthah	1	56	M
Chrystal Norvall	2	30	F
Binni Noonberg	3	35	M
Garland Eatton	4	38	M
Berrie Fougere	5	36	F
Wenda Brubaker	6	71	F
Nan Cornell	7	52	M

Table 2.15: Fragment $\texttt{CUSTOMER}_2$

ID	PROVINCE	LOCATION
0	QC	Dorval
1	MN	Dauphin
2	NU	Iqaluit
3	NL	Corner Brook
4	NS	Halifax
5	QC	Blainville
6	ON	Stratford
7	BC	Kimberley

Vertical fragmentation is considered more complex than horizontal fragmentation. Thus, heuristics are a primary tool to address the problem of optimal vertical fragmentation. Two general heuristic approaches exist: *grouping* and *splitting*. In grouping, initial smaller fragments are combined until certain criteria are reached. In splitting, the process is the reverse – initial larger fragments are split and the relevant criteria is assessed. A more complete overview of the vertical fragmentation can be found in [33].

Finally we note that besides the issue of replication and fragmentation, other important topics of research in the area of distributed databases include distributed

concurrency control and distributed queries. The objective of research in the area of distributed concurrency control focuses on defining policies and methodologies for establishing concurrent utilization of distributed databases by multiple users while preserving the integrity of data and providing for acceptable performance [33]. Work in the area of distributed queries addresses questions about devising rules, as well as approaches for generating queries that can be efficiently executed on the distributed database [33]. An overview of current developments and achievements in the field of distributed databases is presented in the next section.

2.4 Related work

In previous sections we defined the main directions relevant to the development of distributed databases. These can be summarized in the following list:

- Fragmentation
- Replication
- Distributed queries
- Distributed concurrency control

In this section, we elaborate on recent advances in these areas.

2.4.1 Fragmentation

Besides the term fragmentation, the concept of dividing relations into sub-relations is also commonly referred to as partitioning. These two terms are interchangeable; thus, we use both terms below, depending on the source.

The topic of distributed database fragmentation draws considerable attention from researchers. Presently, a number of approaches to the problem of fragmentation have been developed. A thorough overview of advanced partitioning strategies and their implementation is provided in [16]. This work lists, briefly describes, and attempts to evaluate modern advanced approaches to data partitioning, including hash partitioning, range partitioning, non-deterministic strategies and indexed-based partitioning.

One of the novel approaches to data fragmentation is the usage of workload as a determining factor in the fragmentation procedure. In this approach, the workload is analyzed and a fragmentation scheme is proposed so as to minimize cross-partition transactions. This technique is studied in [5], which proposes an approach called Schism, a novel workload-driven, graph-based replication/partitioning strategy. The problem is also examined in [29] and [9]. Both papers define the objective of offering partitioning that minimizes expensive cross-partition transactions, while utilizing information about query workload. The algorithm proposed in [29] relies on the query history in order to optimize partitioning. In contrast, the work in [9] proposes a novel partitioning scheme that allows a set of tables to be to co-partitioned based on given join predicates.

Work in [24] introduces an economic model to automatically balance the supply and demand of data fragments, replicas, and cluster nodes in an adaptive data distribution framework called NashDB. The authors also define the notions of tuple and fragment value, which is used for obtaining optimal fragmentation.

Research performed in [2] postulates that traditional partitioning schemes may not effectively address certain types of applications, such as social networks, that ultimately require what the authors call fine-grained partitioning. In their work, the

authors elaborate on lookup tables that play an essential role in maintaining fine-grained partitioning.

Finally, the authors in [25] present a partitioning *advisor* that recommends the best partitioning design for an expected workload.

2.4.2 Replication

Various works that were mentioned in the section dedicated to fragmentation also elaborate on the topic of replication. This research, including [5], [29] and [24], considers replication and fragmentation as two integral processes underlying distributed database design and thus offers approaches that strive for optimal fragmentation and replication depending on selected factors (e.g. workload).

However, there is a relatively small cluster of work purely focused on replication. The authors of [19] elaborate on two scalable replication techniques that exploit lazy update propagation and workload information. The objective of their work is to guarantee strong consistency in distributed databases while at the same time providing viable performance. Another novel approach to data replication is offered in [15], in which the authors propose an architecture called Asynchronous Parallel Table Replication. This replication architecture distinguishes between non-replicated OLTP workloads and utilizes replication for OLAP workloads.

2.4.3 Distributed queries/transactions

An overview of the current problems related to distributed transactions in modern heterogeneous environments is given in [1]. Additionally, this paper proposes a client-coordinated transaction commitment protocol that does not rely on a central coordinating infrastructure. In [11], the authors target the area of query optimization

for distributed databases. In their work they develope new techniques to generate efficient plans for SQL queries involving multiway joins over partitioned tables. A scalable transaction scheduling and data replication layer called Calvin is presented in [3]. The objective of this work is to manage distributed transactions so as to minimize contention costs associated with distributed transactions and to expose no single point of failure.

2.4.4 Distributed concurrency control

An exhaustive overview and evaluation of six classic and modern protocols for concurrency control in distributed databases is given in [23]. The authors in [8] elaborate upon and compare two low overhead concurrency control schemes that provide for effective handling of network stalls. In addition, the work described in [32] proposes an in-memory distributed optimistic concurrency control protocol called Sundial. According to the authors this protocol addresses two major issues in distributed databases – high latency and the high rate of transaction aborts. In order to achieve these goals, the proposed protocol utilizes caching and a methodology for dynamically establishing logical order among transactions at runtime.

2.5 Conclusion

In this chapter, we have emphasized the significance of distributed data in the world of modern databases. We also provided an overview of the primary concepts underlying distributed databases, such as fragmentation, replication, distributed transactions and distributed concurrency control. Finally, we described the most notable advances in research in the area of of distributed databases. In the following chapters, we will

discuss the application of various principles of distributed databases, using the open source Sequoia DB as our implementation target.

Chapter 3

RAIDb-3: a flexible, commodity based distributed database

3.1 Introduction

The current research examines and develops opportunities for database clustering and distribution by building upon the foundation of the Sequoia DB. Sequoia is database cluster middleware [27] written in Java and provided via an open source license. It uses JDBC for transparent access to the database cluster, as well as for communicating with database nodes within the cluster. As such, it can be utilized with any existing JDBC-compatible DBMS and client software. Figure 3.1 illustrates the core model.

3.2 HSQLDB - a robust, high-performance DBMS

In previous chapters we mentioned that distributed systems utilize database nodes as building blocks in creating distinct data distribution models in which each individual node holds full or partial views of the database. The individual database nodes can be implemented within the distributed database framework as a vendor specific solution. However, it can be more practical and cost-effective to utilize traditional,

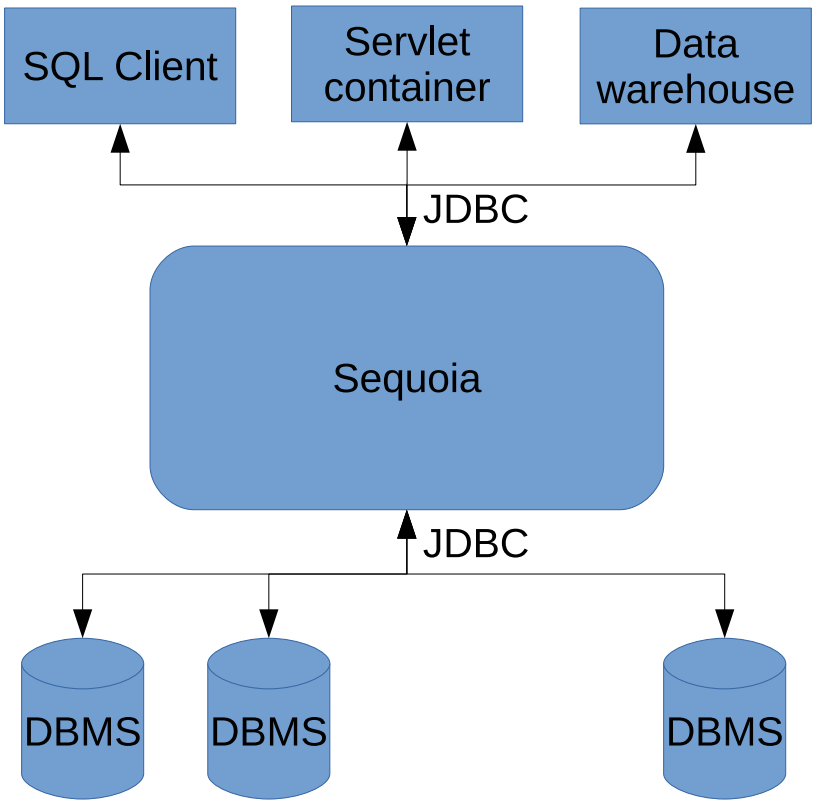

Figure 3.1: Sequoia framework overview

well-evaluated, and well-tested stand-alone DBMSes as the back-end database nodes. The selection of the specific DBMS implementation greatly depends on the communication interfaces to be used between the distributed system core and database nodes.

In our case, Sequoia utilizes JDBC as the primary communication interface. Presently, there is a variety of DBMS implementations offering a JDBC interface, including MySQL, Oracle, PostgreSQL and HSQLDB. We analyzed available commodity of DBMSes and, for this research, we have chosen HSQLDB [13].

HSQLDB (Hyper SQL Database) is a relational database management system written in Java and available under a BSD license, meaning that its source code is open. There are a number of reasons why we selected HSQLDB:

- *HSQLDB is written in Java:* as such, there is no intermediate layer of translation between native code and the Java API.
- *HSQLDB offers high performance:* as per evaluation results available on the HSQLDB website [13], HSQLDB consistently shows superior performance relative to JDBC/MySQL, JDBC/Apache Derby.
- *HSQLDB supports in-memory databases:* placing databases wholly in memory is a novel solution that often produces increased performance and avoids drawbacks associated with the utilization of physical disks.

3.3 Sequoia's original design

Sequoia consists of two principal components – a JDBC driver and the Sequoia controller. The JDBC driver is loaded by JDBC-compatible client software and supports the transfer of SQL requests and the subsequent distribution of data. The controller, in turn, processes requests received from clients, performs load balancing in accordance with a user-defined clustering scheme, and executes queries on the DBMS.

In principle, the Sequoia controller can be used with any JDBC-compatible DBMS, thereby supporting heterogeneous DBMS clusters. However, in the current research we do not explore this feature, nor the option of combining several distinct Sequoia controllers into a network of distributed Sequoia clusters.

In any case, the main purpose of the Sequoia controller is to host a *virtual database*. In short, a virtual database is a software abstraction of the underlying cluster of DBMSes. An external client connecting to Sequoia sees the virtual database as a "regular" DBMS – this point is in fact crucial for deploying Sequoia within an existing SQL-based infrastructure. Note that one Sequoia controller may host multiple

virtual databases. Figure 3.2 illustrates the structure of the controller and its various supporting elements [27].

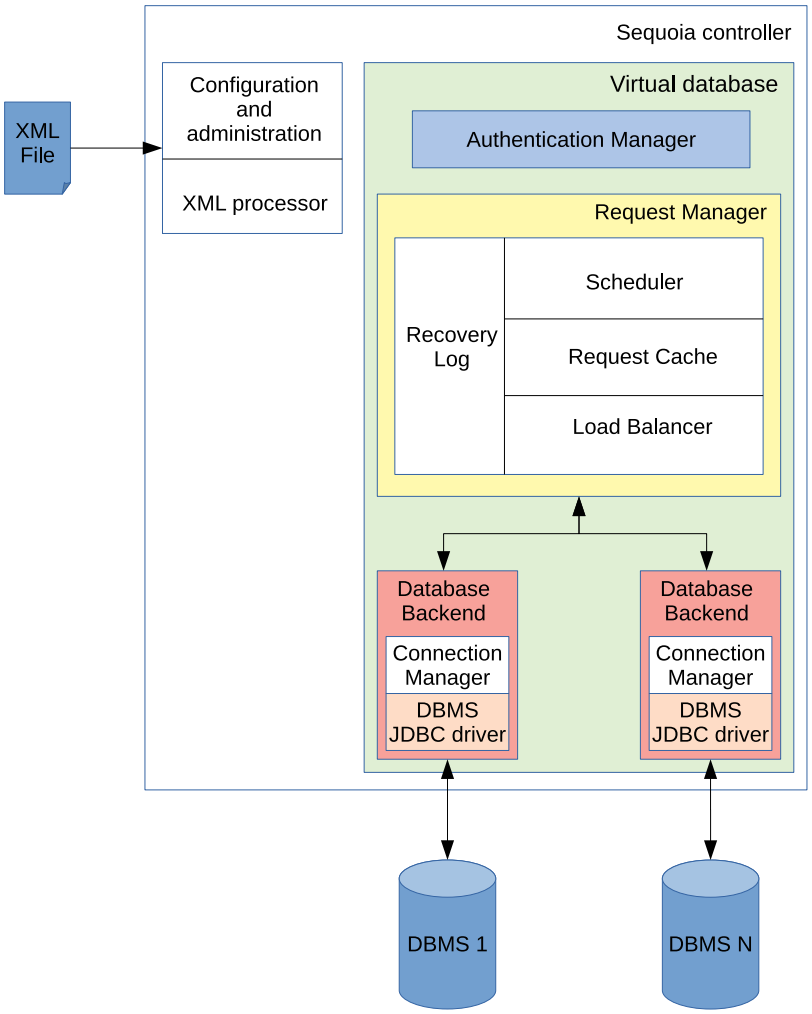

Figure 3.2: Components of a virtual database within the Sequoia controller

Core components include:

- *Configuration and administration:* provides for configuration of the Sequoia

controller and virtual database by means of editable XML files and allows remote administration through a physical console.

- *Authentication manager:* performs verification of virtual database login/password sessions.

- *Request manager:* processes SQL queries received by the JDBC driver from external SQL clients. It, in turn, consists of the following components:

  - *Scheduler:* this component performs scheduling of incoming requests in accordance with the selected clustering RAIDb level.

  - *Request caches:* optional components that perform caching of parsed SQL queries, query metadata and query results. In the current research we do not use this facility.

  - *Load balancer:* this component distributes the query load to the underlying back-ends in accordance with the selected RAIDb level.

  - *Recover log:* an optional component that allows back-ends to dynamically recover from failure. This component is not utilized in the current work.

- *Database back-end:* handles connectivity with the physical back-end DBMS. It loads the JDBC driver and processes requests and data exchange through the Connection Manager.

3.4 Sequoia RAIDb

3.4.1 Overview

As mentioned earlier, RAIDb stands for a Redundant Array of Inexpensive Databases. The primary objective is the achievement of scalability, high availability, superior performance and improved fault tolerance relative to individual databases. At the same time, the RAIDb concept implies that the implementation hides all distribution-related complexity and provides the database client with an abstraction of a single traditional database.

Similar to the more familiar Redundant Array of Inexpensive Disks (RAID), RAIDb distinguishes between several distinct levels of functionality. Below, we provide a brief overview of Sequoia's *original* RAIDb levels.

3.4.2 RAIDb-0

RAIDb-0, as depicted in Figure 3.3, offers distribution of complete tables among back-end nodes. In other words, the same table may not be distributed among several nodes. This level requires at least two back-end nodes and offers moderate performance scalability. That being said, RAIDb-0 does not provide fault tolerance, as tables are not replicated in any way.

3.4.3 RAIDb-1

In contrast, RAIDb-1, depicted in Figure 3.4, offers full database replication among back-ends. This level offers increased fault-tolerance, though it can not provide any performance gain since queries are still executed against a full copy of a single database.

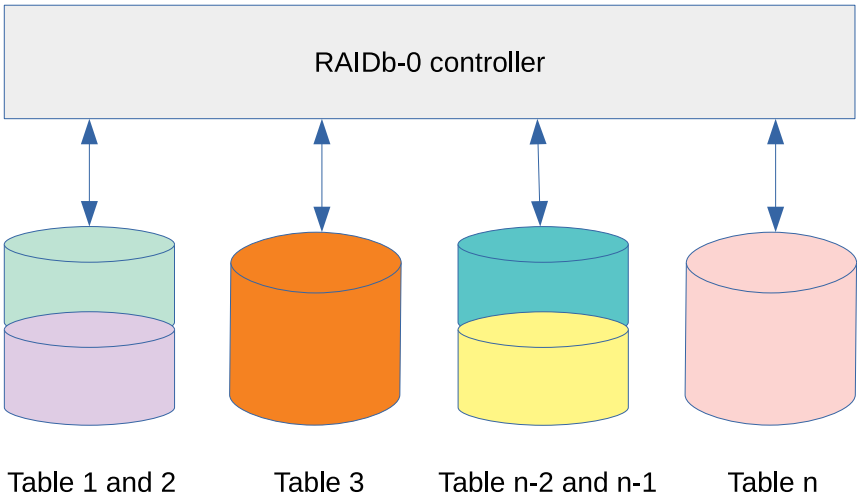

Figure 3.3: RAIDb-0 structure

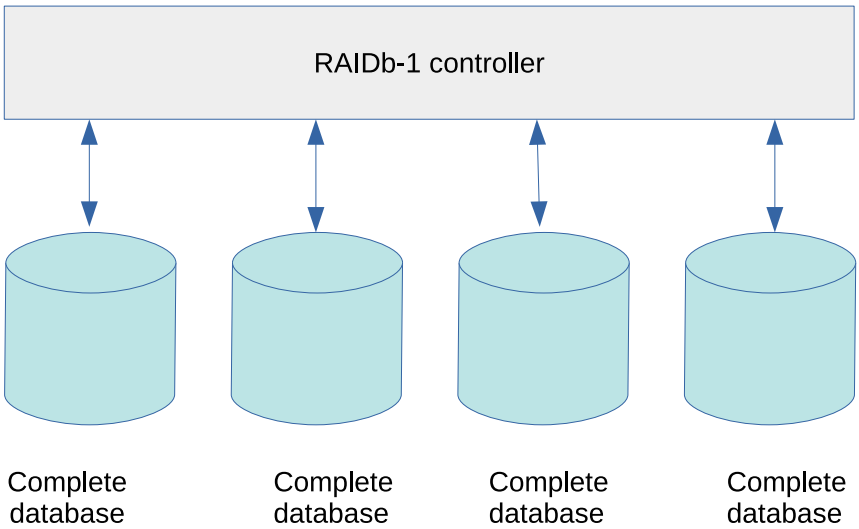

Figure 3.4: RAIDb-1 structure

3.4.4 RAIDb-2

RAIDb-2, illustrated in Figure 3.5, offers partial replication and is an intermediate solution between RAIDb-0 and RAIDb-1. In RAIDb-2, complete tables of the same database can reside on various back-ends. Moreover, each database table must be replicated at least once. In doing so, RAIDb-2 combines the improved performance of RAIDb-0 with the improved fault-tolerance of RAIDb-1.

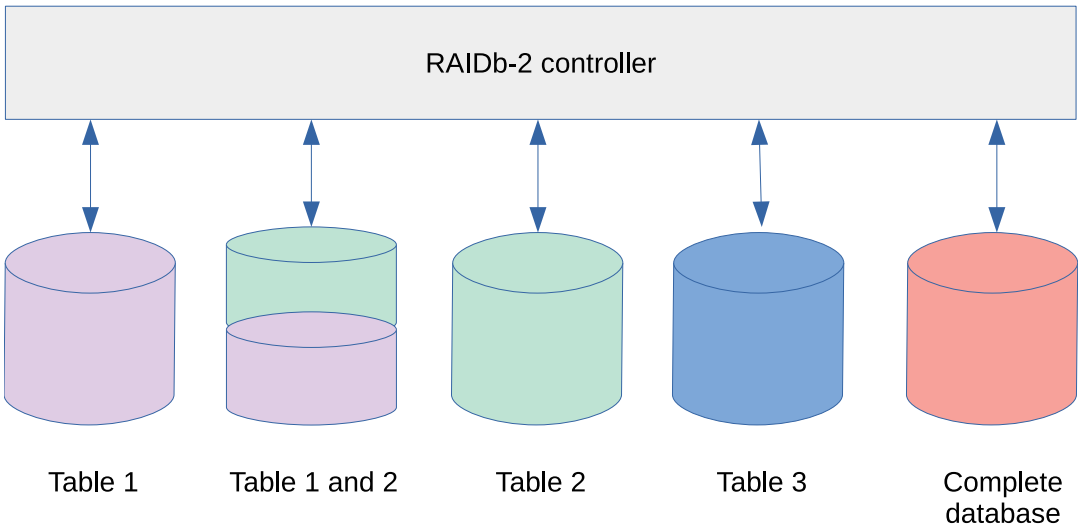

Figure 3.5: RAIDb-2 structure

3.4.5 Other RAIDb schemes

Besides the RAIDb levels mentioned above, Sequoia provides for *nested* RAIDb schemes in which one RAIDb cluster can be embedded as a node in another RAIDb cluster. We do not consider such models in the current research, as it is not relevant to our target research program.

3.5 Research objective

In this work, we investigate new distribution schemes (within Sequoia), with the aim to provide enhanced performance and scalability. To limit the scope of the research we will not consider fault-tolerance in this thesis. We note, however, that fault-tolerance can be achieved through the usage of local back-end RAID constructs. The general application and options for RAID are examined in more detail in [21].

In the previous section, we briefly examined RAIDb levels available in Sequoia. These levels offer either partial distribution of a virtual database among database back-ends or no distribution at all. In the case of partial distribution, such as RAIDb-0 and RAIDb-2, distribution is only possible using *table granularity* (i.e., any database back-end containing a given table must hold the complete table). Furthermore, RAIDb-2 requires the existence of at least one back-end holding the complete database.

The limitations of the original Sequoia design became a starting point for the current research in that we were interested in designing a data model in which distribution is accomplished at the granularity of *table rows*. For simplicity, we call this distribution model RAIDb-3 (note that there is no conceptual correlation with RAID3). Its base structure is depicted in Figure 3.6.

As the illustration suggests, we can expect that a RAIDb-3 model should offer increased performance and scalability due to the higher degree of distribution. How well this assumption correlates with experimental data we will examine later.

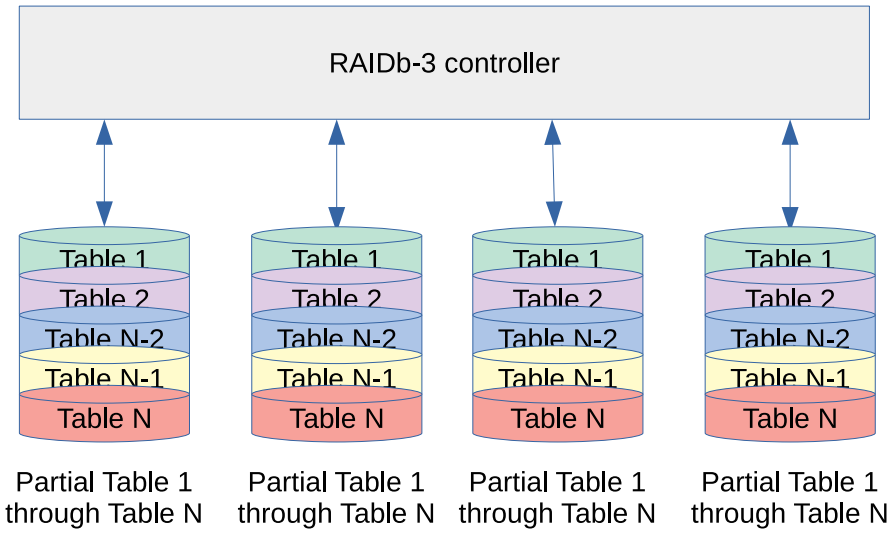

Figure 3.6: RAIDb-3 structure

3.6 RAIDb-3 design considerations

3.6.1 Introduction

In the current context, we may divide the data operations performed on conventional DBMSes into two groups:

1. Data population
2. Data retrieval or querying

Of course, the standard SQL language offers a considerably broader set of operations on relational databases (e.g., manipulating schemas [13]). However, such functionality is not relevant to this research program.

Data retrieval can be further divided into two basic forms:

1. Non-join query
2. Join-based query

Typically, join queries are performed by exploiting both primary and foreign keys [13], though this is not a mandatory requirement. In our research, we will only consider join queries that do in fact utilize primary and foreign keys. As such, we may primarily focus on RAIDb-3 data model principles that can then be extrapolated to all possible types of joins, and not on the implementation complexity that is associated with non-primary/foreign key joins. Here, by implementation complexity we imply not only the complexity of the architectural and/or data model but the complexity of the supporting subsystems, such as the query parser. We note that we will come back to the case of the more complex queries in the Future Work section.

Below, we fully describe the concept of the RAIDb-3 cluster developed in the course of this research. We will build on the data manipulation patterns presented in this section.

3.6.2 Data population

Data population in SQL DBMSes essentially includes two phases:

1. Table creation

2. Filling tables with rows containing data

As we mentioned earlier, full horizontal partitioning - on which our RAIDb-3 is based - splits tables so that its rows are distributed among nodes. Hence, in order to accommodate this principle, there should be a number of database back-ends that contain tables with a structure compatible with that of the original table. Here, by "compatible structure", we are referring to the number and type of columns (database back-end tables may also contain additional internal data values). In order to simplify the implementation of our RAIDb-3 model, a compatible instance of the original table

is created in every database back-end. Instances created in the back-ends are not an exact copy of the original table due to the presence of internal auxiliary data. However, the representation of the table in the RAIDb-3 virtual database completely reflects the structure of the original table.

For example, let's assume that an SQL client connects to Sequoia and creates the table `CUSTOMER` by issuing the statement shown in Listing 3.1.

Listing 3.1: Creating table `CUSTOMER`

```
CREATE TABLE Customer(CustomerId INTEGER GENERATED ALWAYS AS
    IDENTITY PRIMARY KEY, CustomerName VARCHAR(64),
    CustomerAge INTEGER, CustomerSex CHAR(1),
    CustomerProvince CHAR(2), CustomerLocation VARCHAR(64));
```

In this case, the RAIDb-3 cluster processes the statement and creates a compatible `CUSTOMER` table in every active database back-end.

In order to handle join queries, the RAIDb-3 model must perform additional processing on the original `CREATE TABLE` statement. Specifically, it verifies whether the statement references primary and foreign keys. If primary and/or foreign key(s) are found in the client request, the RAIDb-3 controller creates a one-column table(s) – a *primary key table* if the primary key is referenced in the statement and a *foreign key table* if a foreign key is identified. In practice, there can be multiple columns within a primary or foreign key. However, for the sake of simplicity, we do not consider such situations. In any case, these one-column tables are created in an internal database that is physically separate from the data nodes. We discuss the structure and motivation for this *Key Database* in Section 3.6.3.

After the table is successfully created, an SQL client may start appending data to the table by inserting rows. For our RAIDb-3 model the crucial issue at this

step is whether the table contains primary or foreign keys. If a table has neither primary nor foreign keys, rows are inserted into the database back-end using a simple *round-robin* algorithm. No further operations need to be performed. We chose a round-robin algorithm because it offers even distribution of data among database beck-ends. However, in theory, this algorithm might be extended or modified; for example, by assigning variable weights to database back-ends.

We demonstrate this logic in the following example. We consider the `CUSTOMER` table represented in Table 3.1.

Table 3.1: View of virtual database for table `CUSTOMER`

CUST. ID	CUSTOMER NAME	CUST. AGE	CUST. SEX	CUST. PROVINCE	CUSTOMER LOCATION
0	Jen Nemhauser	38	M	QC	Dorval
1	Sydel Jephthah	56	M	MN	Dauphin
2	Chrystal Norvall	30	F	NU	Iqaluit
3	Binni Noonberg	35	M	NL	Corner Brook
4	Garland Eatton	38	M	NS	Halifax
5	Berrie Fougere	36	F	QC	Blainville
6	Wenda Brubaker	71	F	ON	Stratford
7	Nan Cornell	52	M	BC	Kimberley

In addition, we extend the example with a RAIDb-3 cluster containing three database back-ends. Table `CUSTOMER` is created using a `CREATE TABLE` SQL statement

with the form depicted in Listing 3.2.

Listing 3.2: Creating table `CUSTOMER` in back-end

```
CREATE TABLE Customer(CustomerId INTEGER, CustomerName
    VARCHAR(64), CustomerAge INTEGER, CustomerSex CHAR(1),
    CustomerProvince CHAR(2), CustomerLocation VARCHAR(64));
```

In the course of processing this statement, our RAIDb-3 implementation creates a `CUSTOMER` table in every active database back-end. The table is then populated using a series of `INSERT` statements, which have the general pattern shown in Listing 3.3:

Listing 3.3: Generic `INSERT` statement

```
INSERT INTO Customer VALUES(0, 'Jen␣Nemhauser', 38, 'M',
    'QC', 'Dorval');
```

On completion of all the INSERT operations, the individual database back-ends have the data fragments listed in Table 3.2 - Table 3.4.

Table 3.2: View of database back-end #1 for table `CUSTOMER`

CUST. ID	CUSTOMER NAME	CUST. AGE	CUST. SEX	CUST. PROVINCE	CUSTOMER LOCATION
0	Jen Nemhauser	38	M	QC	Dorval
3	Binni Noonberg	35	M	NL	Corner Brook
6	Wenda Brubaker	71	F	ON	Stratford

However, if the original table contains primary and/or foreign key(s), an additional step is dynamically appended to the procedure described above. Specifically, the values of primary and/or foreign key(s) are saved in supporting tables in the dedicated

Table 3.3: View of database back-end #2 for table `CUSTOMER`

CUST. ID	CUSTOMER NAME	CUST. AGE	CUST. SEX	CUST. PROVINCE	CUSTOMER LOCATION
1	Sydel Jephthah	56	M	MN	Dauphin
4	Garland Eatton	38	M	NS	Halifax
7	Nan Cornell	52	M	BC	Kimberley

Table 3.4: View of database back-end #3 for table `CUSTOMER`

CUST. ID	CUSTOMER NAME	CUST. AGE	CUST. SEX	CUST. PROVINCE	CUSTOMER LOCATION
2	Chrystal Norvall	30	F	NU	Iqaluit
5	Berrie Fougere	36	F	QC	Blainville

RAIDb-3 database. We illustrate this step below.

As in the previous example, we consider a RAIDb-3 cluster consisting of three database back-ends. In addition to the `CUSTOMER` table illustrated in Table 3.1, we also include the `INVOICE` table listed in Table 3.5.

The statement provided in Listing 3.4 is used to create the `INVOICE` table.

Listing 3.4: Generic `INSERT` statement

```
CREATE TABLE Invoice(InvoiceId INTEGER GENERATED ALWAYS AS
    IDENTITY PRIMARY KEY, InvoiceCustomerId INTEGER FOREIGN
    KEY REFERENCES Customer(CustomerId), InvoiceTotal DOUBLE,
    InvoiceTimeStamp TIMESTAMP);
```

To populate the `INVOICE` table in the virtual database we employ a set of SQL INSERT statements that use the pattern depicted in Listing 3.5.

Table 3.5: View of virtual database for the `INVOICE` table

INVOICEID	INVOICE CUSTOMERID	INVOICETOTAL	INVOICETIMESTAMP
0	4	91.89	2016-02-04 09:32:58
1	6	24.81	2017-07-20 20:34:20
2	0	122.63	2008-07-19 23:02:56
3	1	128.8	2016-02-07 14:55:54
4	3	56.25	2015-07-20 09:34:30
5	5	100.00	2014-08-18 08:40:40
6	1	25.50	2017-10-13 12:35:50
7	6	63.40	2017-02-03 15:10:10
8	4	89.66	2016-01-06 14:24:20

Listing 3.5: Generic statement to populate `INVOICE`

```
INSERT INTO Invoice VALUES(DEFAULT, 0, 91.89, '2016-02-04␣
    09:32:58');
```

Here, the placeholder `DEFAULT` is used to allow Sequoia to automatically generate the values of the primary key. We will return to the topic of primary key generation and usage in the next section, which is dedicated to a presentation of the Key Database.

Similar to the case of the the `CUSTOMER` table, the `INVOICE` table is horizontally partitioned among database back-ends as shown in Table 3.6 - Table 3.8.

Table 3.6: View of database back-end #1 for table `INVOICE`

INVOICEID	INVOICE CUSTOMERID	INVOICETOTAL	INVOICETIMESTAMP
0	4	91.89	2016-02-04 09:32:58
3	1	128.8	2016-02-07 14:55:54
6	1	25.50	2017-10-13 12:35:50

Table 3.7: View of database back-end #2 for table `INVOICE`

INVOICEID	INVOICE CUSTOMERID	INVOICETOTAL	INVOICETIMESTAMP
1	6	24.81	2017-07-20 20:34:20
4	3	56.25	2015-07-20 09:34:30
7	6	63.40	2017-02-03 15:10:10

We note the existence of specific columns in `CUSTOMER` and `INVOICE` (we prepend the corresponding table name to the name of the column and use '.' as a delimiter):

- `Customer.CustomerID`: assigned a primary key constraint, as per Listing 3.1

Table 3.8: View of database back-end #3 for table `INVOICE`

INVOICEID	INVOICE CUSTOMERID	INVOICETOTAL	INVOICETIMESTAMP
2	0	122.63	2008-07-19 23:02:56
5	5	100.00	2014-08-18 08:40:40
8	4	89.66	2016-01-06 14:24:20

(note that additional syntax is HSQLDB-specific [13]).

- `Invoice.InvoiceId:` assigned a primary key constraint, as per Listing 3.4 (again, additional syntax is HSQLDB-specific).

- `Invoice.InvoiceCustomerId:` designated as a foreign key that references the primary key `CustomerId` in the `CUSTOMER` table, as per Listing 3.4.

Considering that both tables contain primary or foreign (or both) key constraint(s), additional tables are created in the dedicated internal Key Database as follows:

- `CustomerPRM:` for holding primary key values from the `CUSTOMER` table - shown in Table 3.9.

Table 3.9: View of the Key Database for `CustomerPRM`

PRIMARY_KEY
0
1
2
3
4
5
6
7

- `InvoicePRM`: for holding primary key values from table `INVOICE` - shown in Table 3.10

Table 3.10: View of the Key Database for `InvoicePRM`

PRIMARY_KEY
0
1
2
3
4
5
6
7
8

- `InvoiceFRG`: for holding foreign key values from `INVOICE` - shown in Table 3.11.

Table 3.11: View of the Key Database for table `InvoiceFRG`

FOREIGN_KEY
4
6
0
1
3
5
1
6
4

3.6.3 Role of the Key Database

In the previous section, we briefly mentioned the Key Database that is used for cases in which the primary and/or foreign key(s) are defined within the associated tables. In this section we elaborate on the topic of the Key Database.

Our RAIDb-3 implementation offers row-level distribution granularity among database back-ends. We showed how data is populated among back-ends in the previous section. However, the process of retrieving data from the database back-ends, when row granularity is required, is not trivial. For the simpler case of non-join queries the original query can be propagated down to every database back-end and the partial result data can then be aggregated into a final result.

Such an approach will not work for join queries - which we describe in the next section - as a retrieval query of this form issued against tables in which rows are distributed among different database back-end would emit an incorrect result. For example, we can consider a join operation between tables `CUSTOMER` (Table 3.1) and `INVOICE` (Table 3.5) with a join predicate establishing a relationship between primary and foreign key as follows: `CUSTOMER ID=INVOICE CUSTOMER ID`. The result of this JOIN is depicted in 3.12.

Table 3.12: Result of properly joining `INVOICE` and `CUSTOMER`

INV. ID	INV. CUS. ID	INV. TOTAL	INV. TIMESTAMP	CUST. ID	CUST. NAME	CUST. AGE	CUST. SEX	CUST. PROV.	CUST. LOCATION
0	4	91.89	2016-02-04 09:32:58	4	Garland Eatton	38	M	NS	Halifax
3	1	128.8	2016-02-07 14:55:54	1	Sydel Jephthah	56	M	MN	Dauphin
6	1	25.50	2017-10-13 12:35:50	1	Sydel Jephthah	56	M	MN	Dauphin
1	6	24.81	2017-07-20 20:34:20	6	Wenda Brubaker	71	F	ON	Stratford
4	3	56.25	2015-07-20 09:34:30	3	Binni Noonberg	35	M	NL	Corner Brook
7	6	63.40	2017-02-03 15:10:10	6	Wenda Brubaker	71	F	ON	Stratford
2	0	122.63	2008-07-19 23:02:56	0	Jen Nemhauser	38	M	QC	Dorval
5	5	100.00	2014-08-18 08:40:40	5	Berrie Fougere	36	F	QC	Blainville
8	4	89.66	2016-01-06 14:24:20	4	Garland Eatton	38	M	NS	Halifax

If, on the other hand we execute this join between *fragments* of the `CUSTOMER` and `INVOICE` tables, for instance with the fragments stored in database back-end #2 (Table 3.3 and Table 3.7), the result will be represented by the empty set, as there are no values in column `CUSTOMER_ID` of the `CUSTOMER` table that match any values of the column `INVOICE_CUSTOMER_ID` in the `INVOICE` table. This is not the case of course when the full tables are utilized. However in the distributed database, this empty set represents one individual partial result which is subsequently aggregated by the RAIDb-3 controller with partial results obtained from the other database back-ends, to produce a final complete result (the topic of data retrieval is discussed in the next section). This final result will be invalid, as the partial result described above excludes a number of records from the final aggregated result, which otherwise would be present in the result of the join executed on the original `CUSTOMER` and `INVOICE` tables. To illustrate this outcome, the aggregated result of joining all fragments of the `CUSTOMER` and `INVOICE` tables is shown in Table 3.13. It should be self-evident that this result is completely inconsistent with the correct result shown in Table 3.12. In fact, in the absence of explicit support, a join query may only be issued against tables that contain a complete set of rows in one place (i.e., the tables must completely reside within one back-end).

Table 3.13: Aggregated result when joining fragmented `INVOICE` and `CUSTOMER`

INV. ID	INV. CUS. ID	INV. TOTAL	INV. TIMESTAMP	CUST. ID	CUST. NAME	CUST. AGE	CUST. SEX	CUST. PROV.	CUST. LOCATION
5	5	100.00	2014-08-18 08:40:40	5	Berrie Fougere	36	F	QC	Blainville

Here we arrive at a central feature of the current research – the use of an internal

dedicated database to store tables that contain information on columns participating in join queries. As noted previously, we specifically limit these columns to primary and foreign keys. In doing so, we are able to support row granularity while at the same time maintaining one of the primary features of any DBMS system (i.e., basic join operations).

With this model, the actual join query is executed on the Key Database tables only. Results obtained during execution of this supplemental query are then used to form non-join queries executed against all active database back-ends. Ultimately, intermediate results are combined and delivered to the SQL client. Note that data retrieval is examined in detail in the next section.

The proposed RAIDb-3 model has an important implication on data arrangement within the database back-ends. Specifically, the primary key for each table must be unique across all active database back-ends. In other words, the primary key must be assigned globally by Sequoia. This functionality is realized via the direct exploitation of the Key Database.

As a concrete example, let us assume that the RAIDb-3 controller receives an `INSERT` statement for a table containing a primary key, such as the query shown in Listing 3.6.

Listing 3.6: Example of `INSERT` statement with primary key

```
INSERT INTO Invoice VALUES(DEFAULT, 0, 91.89, '2016-02-04␣
    09:32:58');
```

Sequoia expects the value of the primary key to be filled with the placeholder `DEFAULT` (otherwise an error is returned). The controller then generates an `INSERT` statement addressed to the Key Database table – in our case `InvoicePRM`. The specific

syntax of this statement forces the DBMS to generate a unique table-wide value for the primary key in the inserted row. As soon as the row is inserted into the `InvoicePRM` table in the Key Database, the value of the auto-generated primary key is fetched and then used as an explicitly specified primary key value in the `INSERT` statement that is generated for the currently selected database back-end (as per the round-robin distribution). The details of this procedure will be covered in the implementation section.

In summary, the Key Database plays a crucial role in the realization of the RAIDb-3 model. It handles two essential tasks:

1. Storage of primary and foreign keys in the related column tables
2. Generation of unique primary keys

With the introduction of the concept of the Key Database, the updated design structure of Sequoia assumes the form shown in Figure 3.7. Here, the Key Database is embedded within the Key Manager. Besides managing the Key Database, this component also handles several miscellaneous functions that will be covered in the implementation section.

3.6.4 Data retrieval

The standard approach to retrieve data from a relational database is to use a `SELECT` statement [13]. Furthermore, as mentioned earlier, `SELECT` queries can be divided into two basic forms: Join and Non-join. Join operations are in fact the most difficult part of query processing within the RAIDb-3 model. This is due to the fact that if join queries are executed locally on database back-ends and intermediate results are then

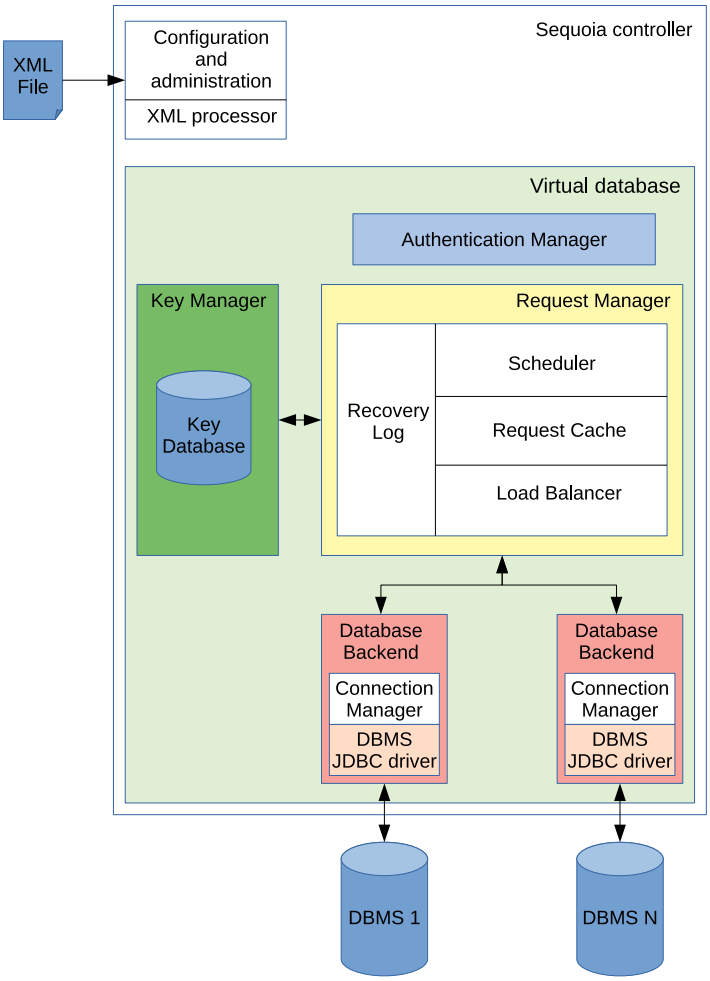

Figure 3.7: Components of virtual database within the Sequoia controller, with RAIDb-3 extension.

directly combined into a final result, this final set is likely to be invalid, as records to be joined may be located on different database back-ends. This shortcoming was discussed in Section 3.6.3.

3.6.4.1 Non-join operations

Since a non-join query implies the absence of any dependency between columns in different tables, records can be retrieved directly from the database back-ends and then combined into a final result. In our RAIDb-3 model, we utilize an approach in which we first identify the query logic in the `WHERE` clause of the `SELECT` statement and then use it to fetch records. Note that the `WHERE` clause is associated with the *selection* operation in relational algebra. In general, DBMS systems attempt to perform the selection operation first because, by doing so, the query engine immediately creates a smaller intermediate set for the subsequent relational operations.

Hence, in order to process a data retrieval query we extract the logic from the `WHERE` clause of the original `SELECT` statement and use it to generate back-end specific `SELECT` statements in which the column list is replaced with a `'*'` wild-card placeholder. For example, if we consider the Customer table created in Listing 3.2, the original non-join query might assume the form shown in Listing 3.7.

Listing 3.7: Example of `SELECT` statemnt

```
SELECT CustomerProvince, CustomerLocation FROM Customer
    WHERE Customer.CustomerAge>=30 AND
     Customer.CustomerAge<40 AND Customer.CustomerSex='M';
```

Then, the DBMS-generated query issued on all active database back-ends would become the the query depicted in Listing 3.8.

Listing 3.8: `SELECT` statemnt issued on databse back-end

```
SELECT * FROM Customer WHERE Customer.CustomerAge>=30 AND
    Customer.CustomerAge<40 AND Customer.CustomerSex='M';
```

Internal queries for database back-ends are executed in concurrent fashion. Intermediate results are combined in a Sequoia-managed data structure and the final result is pushed to the JDBC driver in accordance with the original query syntax (i.e, as shown in Listing 3.7). An illustration of how intermediate results are combined is given in the next section.

3.6.4.2 Join operations

Processing of join queries involves a number of additional steps. On arrival of the join query, the RAIDb-3 controller identifies join columns in the query. It then constructs an appropriate join query for execution on the Key Database. In order to illustrate the processing logic of the join query mechanism, we will use the previously defined tables `CUSTOMER` (Table 3.1) and `INVOICE` (Table 3.5). For convenience, we illustrate their creation procedures in Listing 3.9 and 3.10 respectively.

Listing 3.9: Creation of table of `CUSTOMER`

```
CREATE TABLE Customer(CustomerId INTEGER GENERATED ALWAYS AS
    IDENTITY PRIMARY KEY, CustomerName VARCHAR(64),
    CustomerAge INTEGER, CustomerSex CHAR(1),
    CustomerProvince CHAR(2), CustomerLocation VARCHAR(64));
```

As per the discussion of the data population process in Section 3.6.2, the following additional column tables are created inside the Key Database for each of the original tables - Table 3.9, Table 3.10 and Table 3.11 (note that the square brackets indicate

Listing 3.10: Creation of table of `INVOICE`

```
CREATE TABLE Invoice(InvoiceId INTEGER GENERATED ALWAYS AS
    IDENTITY PRIMARY KEY, InvoiceCustomerId INTEGER FOREIGN
    KEY REFERENCES Customer(CustomerId), InvoiceTotal DOUBLE,
    InvoiceTimeStamp TIMESTAMP);
```

the sole column attribute within the table).

- `CUSTOMER: CustomerPRM[PRIMARY_KEY]`
- `INVOICE: InvoicePRM[PRIMARY_KEY], InvoiceFRG[FOREIGN_KEY]`

A join query issued by the client might look like the one in Listing 3.11. (the details of the join predicate are not important):

Listing 3.11: Join query

```
SELECT * FROM Invoice, Customer WHERE
    Invoice.InvoiceCustomerId=Customer.CustomerId AND
    Invoice.InvoiceTotal>=50 AND
    Invoice.InvoiceTimeStamp>='2016-02-01 00:00:00' AND
    Invoice.InvoiceTimeStamp<='2016-02-28 23:59:59' AND
    Customer.CustomerAge>=30 AND Customer.CustomerAge<40 AND
    Customer.CustomerSex='M' AND
    Customer.CustomerProvince='QC';
```

To answer this query, a join operation initiated by the RAIDb-3 controller is performed between the primary key `CustomerId` of the table `CUSTOMER` and the foreign key `InvoiceCustomerId` of the table `INVOICE`. The controller identifies the key columns `CustomerId` and `InvoiceCustomerId` and generates the Key Database join query as shown in Listing 3.12.

Listing 3.12: Join query in RAIDb-3 cluster

```
SELECT * FROM InvoiceFRG, CustomerPRM WHERE
    InvoiceFRG.FOREIGN_KEY=CustomerPRM.PRIMARY_KEY;
```

After completing the query on the Key Database, we obtain a complete set of primary and foreign keys to be included in the final result. If we again consider our example, then the execution of Statement 3.12 on Table 3.9 and Table 3.11 produces the result illustrated in Table 3.14.

Table 3.14: Result of executing join on tables `CustomerPRM` and `InvoiceFRG`

InvoiceFRG.FOREIGN_KEY	CustomerPRM.PRIMARY_KEY
4	4
6	6
0	0
1	1
3	3
5	5
1	1
6	6
4	4

In the subsequent step, the RAIDb-3 controller retrieves records from the database back-ends. The controller generates SELECTION queries that fetch only those records from database back-end tables whose primary/foreign key values exist in the result set obtained during the previous step.

In terms of our running example, on completion of Statement 3.12, Sequoia receives the table illustrated in Table 3.14 containing two columns – `InvoiceFRG.FOREIGN_KEY` and `CustomerPRM.PRIMARY_KEY`. With this data, the controller then generates two

non-join queries to be executed on all active database back-ends:

Listing 3.13: Database back-end query for table `CUSTOMER`

```
SELECT * FROM Customer WHERE CustomerId IN
    (CustomerPRM.PRIMARY_KEY);
```

Listing 3.14: Database back-end query for table `INVOICE`

```
SELECT * FROM Invoice WHERE InvoiceCustomerId IN
    (InvoiceFRG.FOREIGN_KEY);
```

Here, it is important to note that the statements given in Listing 3.13 and Listing 3.14 are not valid SQL statements but symbolic expressions used to better illustrate the processing logic. In practice, `CustomerPRM.PRIMARY_KEY` and `InvoiceFRG.FOREIGN_KEY` are arrays of values and not a simple expression or clause.

To further illustrate this technique we again consider our running example. As mentioned earlier, Statement 3.13 is executed on fragments of table `CUSTOMER`. In practice, this means that it is executed on the following tables: Table 3.2, Table 3.3 and Table 3.4. The results of execution are given in Table 3.15 - Table 3.17.

Table 3.15: Partial results from database back-end #1 for table `CUSTOMER`

CUST. ID	CUSTOMER NAME	CUST. AGE	CUST. SEX	CUST. PROVINCE	CUSTOMER LOCATION
0	Jen Nemhauser	38	M	QC	Dorval
3	Binni Noonberg	35	M	NL	Corner Brook
6	Wenda Brubaker	71	F	ON	Stratford

Table 3.16: Partial results from database back-end #2 for table `CUSTOMER`

CUST. ID	CUSTOMER NAME	CUST. AGE	CUST. SEX	CUST. PROVINCE	CUSTOMER LOCATION
1	Sydel Jephthah	56	M	MN	Dauphin
4	Garland Eatton	38	M	NS	Halifax

Table 3.17: Partial results from database back-end #3 for table `CUSTOMER`

CUST. ID	CUSTOMER NAME	CUST. AGE	CUST. SEX	CUST. PROVINCE	CUSTOMER LOCATION
5	Berrie Fougere	36	F	QC	Blainville

Similarly, Statement 3.14 is executed on fragments of the table `INVOICE`. Again, in practice, this means that it is executed on the following tables: Table 3.6, Table 3.7 and Table 3.8. The results of execution are given in Table 3.18 - Table 3.20.

Table 3.18: Partial results from database back-end #1 for table `INVOICE`

INVOICEID	INVOICE CUSTOMERID	INVOICETOTAL	INVOICETIMESTAMP
0	4	91.89	2016-02-04 09:32:58
3	1	128.8	2016-02-07 14:55:54
6	1	25.50	2017-10-13 12:35:50

As was the case with the non-join query, the statements in Listing 3.13 and Listing 3.14 are executed concurrently on the database back-ends. Partial results are returned and stored within an internal data structure. Once all partial sets have been collected,

Table 3.19: Partial results from database back-end #2 for table `INVOICE`

INVOICEID	INVOICE CUSTOMERID	INVOICETOTAL	INVOICETIMESTAMP
1	6	24.81	2017-07-20 20:34:20
4	3	56.25	2015-07-20 09:34:30
7	6	63.40	2017-02-03 15:10:10

Table 3.20: Partial results from database back-end #3 for table `INVOICE`

INVOICEID	INVOICE CUSTOMERID	INVOICETOTAL	INVOICETIMESTAMP
2	0	122.63	2008-07-19 23:02:56
5	5	100.00	2014-08-18 08:40:40
8	4	89.66	2016-01-06 14:24:20

results for the *root* table are aggregated so that a single intermediate table is formed. The aggregation of Table 3.15 - Table 3.17 therefore produces the `CUSTOMER` table depicted in Table 3.21. For the `INVOICE` table, the intermediate result is obtained by aggregating Table 3.18 - Table 3.20, as illustrated in 3.22.

Ultimately, these intermediate tables are joined together and the final results are produced and pushed by the JDBC driver to the client. This final join process is illustrated in Table 3.23, which represents the result of merging Table 3.21 and Table 3.22. While this running example is somewhat involved, and produces a number of intermediate tables, the key point is that Table 3.23 contains the same set of rows as Table 3.12. In other words, the fully distributed RAIDb-3 result on a join query exactly matches the result produced by a single standalone DBMS.

Table 3.21: Intermediate aggregation result for table `CUSTOMER`

CUST. ID	CUSTOMER NAME	CUST. AGE	CUST. SEX	CUST. PROVINCE	CUSTOMER LOCATION
0	Jen Nemhauser	38	M	QC	Dorval
3	Binni Noonberg	35	M	NL	Corner Brook
6	Wenda Brubaker	71	F	ON	Stratford
1	Sydel Jephthah	56	M	MN	Dauphin
4	Garland Eatton	38	M	NS	Halifax
5	Berrie Fougere	36	F	QC	Blainville

Table 3.22: Intermediate aggregation result for table `INVOICE`

INVOICEID	INVOICE CUSTOMERID	INVOICETOTAL	INVOICETIMESTAMP
0	4	91.89	2016-02-04 09:32:58
3	1	128.8	2016-02-07 14:55:54
6	1	25.50	2017-10-13 12:35:50
1	6	24.81	2017-07-20 20:34:20
4	3	56.25	2015-07-20 09:34:30
7	6	63.40	2017-02-03 15:10:10
2	0	122.63	2008-07-19 23:02:56
5	5	100.00	2014-08-18 08:40:40
8	4	89.66	2016-01-06 14:24:20

Table 3.23: Final result after joining intermediate results of `INVOICE` and `CUSTOMER`

INV. ID	INV. CUS. ID	INV. TOTAL	INV. TIMESTAMP	CUST. ID	CUST. NAME	CUST. AGE	CUST. SEX	CUST. PROV.	CUST. LOCATION
0	4	91.89	2016-02-04 09:32:58	4	Garland Eatton	38	M	NS	Halifax
3	1	128.8	2016-02-07 14:55:54	1	Sydel Jephthah	56	M	MN	Dauphin
6	1	25.50	2017-10-13 12:35:50	1	Sydel Jephthah	56	M	MN	Dauphin
1	6	24.81	2017-07-20 20:34:20	6	Wenda Brubaker	71	F	ON	Stratford
4	3	56.25	2015-07-20 09:34:30	3	Binni Noonberg	35	M	NL	Corner Brook
7	6	63.40	2017-02-03 15:10:10	6	Wenda Brubaker	71	F	ON	Stratford
2	0	122.63	2008-07-19 23:02:56	0	Jen Nemhauser	38	M	QC	Dorval
5	5	100.00	2014-08-18 08:40:40	5	Berrie Fougere	36	F	QC	Blainville
8	4	89.66	2016-01-06 14:24:20	4	Garland Eatton	38	M	NS	Halifax

3.7 RAIDb-3 implementation

3.7.1 Introduction

In this section, we cover the implementation specifics of the RAIDb-3 model that has been integrated into Sequoia. As noted, Sequoia is an open source DBMS with a license permitting unlimited modification. Furthermore, Sequoia is written in Java [17] – a mature, well supported high-level object-oriented language that provides for ease of development and debugging. In fact, the Java SDK already contains a variety of data structures and a complete stack of tools for working with JDBC connected databases, including fundamental JDBC driver operations, as well as classes for building and executing SQL statements.

The original Sequoia implementation was designed for JDK 1.6 (Java SE 6). However, in the course of this research project we modified Sequoia so that it could utilize the benefits of JDK 1.8 (Java SE 8). Additionally, the initial Sequoia distribution used a scripted build procedure that we upgraded to a more robust NetBeans [20] project. This allowed simpler project management, convenient code manipulation and efficient debugging and profiling.

Finally, we note that the implementation of RAIDb-3 affected almost every component of the original Sequoia framework depicted in Figure 3.2. Moreover, new components, including the Key Manager, were created in order to extend the core functionality of the original system. A source-code of the Sequoia RAIDb-3 project can be found on `https://sequoiara.sourceforge.io/`.

3.7.2 Key Manager

The Key Manager is a new component introduced into Sequoia as part of the RAIDb-3 implementation. The manager's main responsibility is the manipulation of the Key Database. Additionally, the Key Manager maintains an internal data structure that holds the primary and foreign key column indexes for each table that contains such keys. This information is essential during the processing of the `INSERT` statement.

On Sequoia launch, the Key Manager establishes a connection to the internal Key Database, which is hosted on a dedicated HSDLDB node. It uses the Sequoia infrastructure to accomplish this, obtaining a handle to a JDBC connection of type `java.sql.Connection` [18]. The Key Manager utilizes this handle to execute SQL statements on the Key Database.

3.7.3 Handling `CREATE TABLE` statements

The original statement must first be pre-processed and then modified as necessary. The main purpose of statement pre-processing is to strip out all HSQLDB syntax specifics (e.g., `GENERATED ALWAYS AS IDENTITY`, which is incompatible with the Sequoia statement parser). We perform this step to avoid modifying the Sequoia statement parser itself while, at the same time, preserving the original HSQLDB syntax in the SQL client. This makes it possible to use the same set of test cases for HSQLDB

and for Sequoia.

Upon arrival of the `CREATE TABLE` SQL statement, we first check whether the table to be created contains primary and/or foreign keys. In order to do so, we add a *supplemental statement parser* that ultimately performs statement string analysis and returns indexes of primary and foreign keys.

This functionality is provided by two methods within Sequoia's `AbstractWriteRequest` class:

- `public int GetPkPos()`: parses the statement and fetches the index of the primary key column, if it exists.
- `public int GetFkPos()`: parses the statement and fetches the index of the foreign key column, if it exists (in this research we only consider the case of a single foreign key; however our prototype can also be extended to support multiple foreign keys without changing the architecture).

For example, assume we have the statement shown in Listing 3.15.

Listing 3.15: Arbitrary `CREATE TABLE` statement

```
CREATE TABLE Invoice(InvoiceId INTEGER GENERATED ALWAYS AS
    IDENTITY PRIMARY KEY, InvoiceCustomerId INTEGER FOREIGN
    KEY REFERENCES Customer(CustomerId), InvoiceTotal DOUBLE,
    InvoiceTimeStamp TIMESTAMP);
```

In this case, we receive the following results after calling `GetPkPos()` and `GetFkPos()`:

```
GetPkPos() --> 1;

GetFkPos() --> 2;
```

These operations are encapsulated within the Key Manager method `CreatePkFk`:

```
public void CreatePkFk(AbstractWriteRequest aRequest) throws SQLException
```

In cases where the original statement contains either a primary or foreign key, (i.e. either call to `GetPkPos()` and `GetFkPos()` returns a positive non-zero integer), the Key Manager performs additional operations to store these indexes and to create appropriate tables in the Key Database. Below we consider both scenarios.

1. Statement contains primary key - `GetPkPos()>0`.

   Here, the Key Manager prepares the SQL statement to be executed on the Key Database, as shown in Listing 3.16.

   Listing 3.16: Key Manager `CREATE TABLE` statement for primary key

   ```
   CREATE TABLE name_of_the_tablePRM(primary_key INTEGER
       GENERATED ALWAYS AS IDENTITY PRIMARY KEY);
   ```

   In this case, `name_of_the_table` is the name of the table extracted from the original `CREATE TABLE` statement and suffix `PRM` – standing for "primary" - is appended to this name. Special syntax specific to HSQLDB [13] is used in order to automatically generate the value of the primary key during the `INSERT` operation. The statement is then executed on the Key Database or, to be more precise, on the `Connection` handle to the Key Database (with the help of the Java JDBC classes [18]). This is simply executed as: `Statement.executeUpdate()`

2. Statement contains foreign key – `GetFkPos()>0`.

   In this second case, the Key Manager prepares the SQL statement depicted in Listing 3.17 to be executed on the Key Database:

Listing 3.17: Key Manager `CREATE TABLE` statement for foreign key

```
CREATE TABLE name_of_the_tableFRG(foreign_key INTEGER);
```

Similar to the case of the primary key, the name of the table is extracted from the original `CREATE TABLE` statement and the suffix `FRG` – standing for "foreign" - is appended to the name string. Ultimately, the statement is executed on the Key Database in the same manner as that of the primary key.

Thus, for the sample statement listed in Listing 3.15, the Key Manager would prepare and execute the statements shown in Listing 3.18 and Listing 3.19.

Listing 3.18: Key Manager `CREATE TABLE` statement for primary key

```
CREATE TABLE InvoicePRM(primary_key INTEGER GENERATED ALWAYS
    AS IDENTITY PRIMARY KEY);
```

Listing 3.19: Key Manager `CREATE TABLE` statement for foreign key

```
CREATE TABLE InvoiceFRG(foreign_key INTEGER);
```

In terms of primary and/or foreign key indexes, they are stored within the Key Manager class using a hashmap-based data structure [18]. In this case, the table name is used as a key and the primary/foreign key column indexes are stored in the value structure. The usage of a `HashMap` container is ideal in this context as we can guarantee that identical table names do not exist. Lookup time complexity for the standard `HashMap` is `O(1)` in the average case.

After setting up the Key Manager, the RAIDb-3 controller initializes the database back-ends. In order to do so, the Request Manager deletes any constraints (such as primary or foreign keys) from the original `CREATE TABLE`. This is consistent with the general RAIDb-3 design in that values for primary keys are globally assigned by the

Key Manager. This approach prevents unnecessary database consistency checks, as well as allowing the DBMS to use different database back-ends to store tables with foreign keys, as well as the table containing the primary key that the foreign key references. Ultimately, the modified statement is propagated to the SQL driver by the Load Balancer and executed on each data back-end.

Therefore, if we consider the original `CREATE TABLE` statement given in 3.15, the modified statement produced by the Request Manager and propagated to each database back-end by the Load Balancer assumes the following form:

Listing 3.20: Processed statement

```
CREATE TABLE Invoice(InvoiceId INTEGER, InvoiceCustomerId
    INTEGER, InvoiceTotal DOUBLE, InvoiceTimeStamp TIMESTAMP);
```

To summarize the basic logic of the `CREATE TABLE` mechanism, the workflow of the `CREATE TABLE` statement is depicted in Figure 3.8. From an architectural perspective, we can graphically illustrate the process of handling the RAIDb-3 `CREATE TABLE` statement in Figure 3.9.

3.7.4 Handling the `INSERT` statement

As was the case with `CREATE TABLE`, processing of `INSERT` statements is heavily dependent upon whether the underlying table contains primary and/or foreign keys. In the case where the table from the `INSERT` statement contains neither a primary nor a foreign key, there is no modification of the original statement, and it is passed directly to the Load Balancer component of the RAIDb3 engine, which then propagates the statement to the JDBC driver where it is executed on the appropriate database back-end.

In the design section, we pointed out that records for each table are inserted as

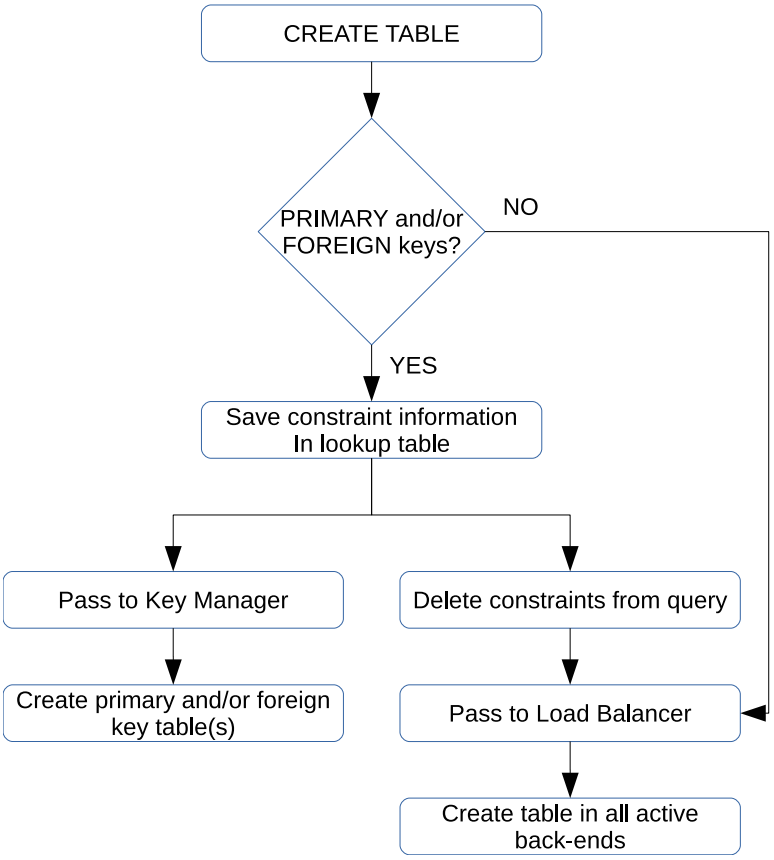

Figure 3.8: The `CREATE TABLE` statement workflow.

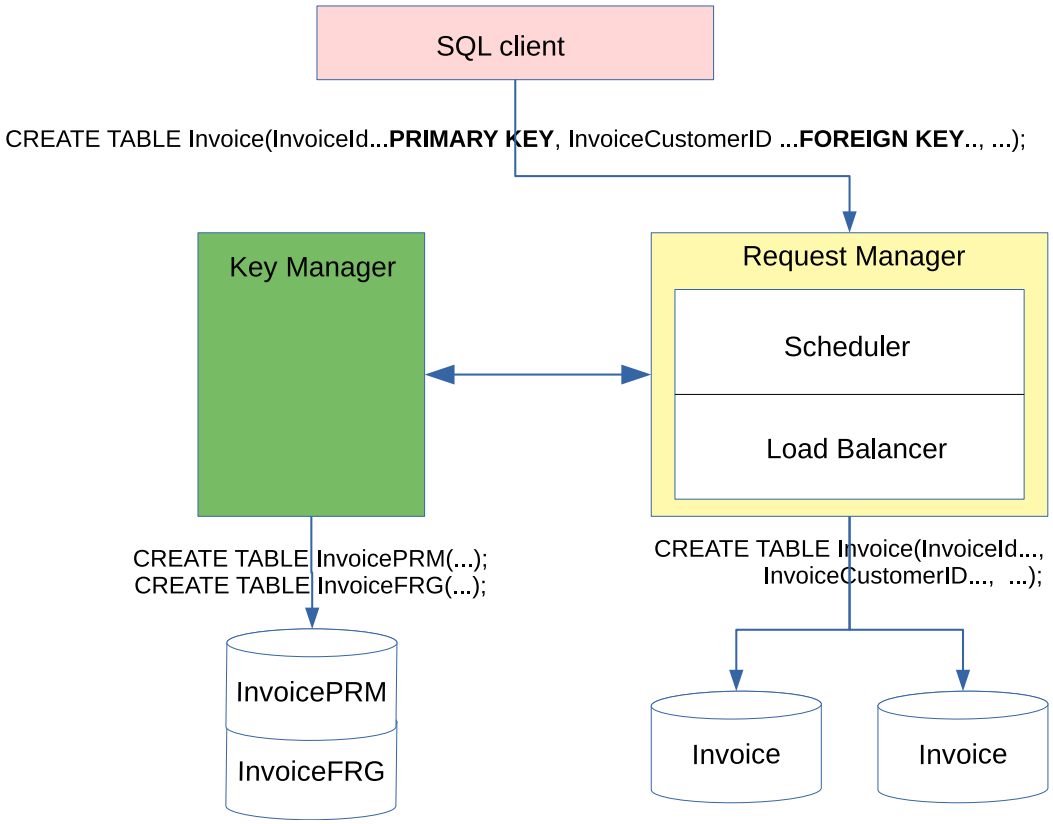

Figure 3.9: The `CREATE TABLE` statement processing overview.

per a round-robin distribution algorithm. This logic is implemented in the RAIDb-3 Load Balancer. The Load Balancer monitors all active back-ends and, more importantly, tracks the back-end that was used last for an `INSERT` operation on a given table. For this purpose the Load Balancer utilizes a data structure based on Java's `ConcurrentHashMap` [18]. In this case, the table name is used as a key and an iterator pointing to the last database back-end involved in the `INSERT` operation is stored in the value structure. Every time the `INSERT` is received, the iterator obtained from `ConcurrentHashMap` is incremented in a circular manner. In this way, the round-robin distribution of records among database back-ends is achieved.

The `ConcurrentHashMap` is a subclass of `HashMap` and in our context provides insert and lookup time complexity of `O(1)` in the average case. In addition to the common benefits of the `ConcurrentHashMap`, we note that it also provides for improved concurrency performance. Specifically, retrieval operations on this data structure do not entail locking [18]. This is quite important considering that `INSERT` statements are executed in concurrent fashion on the database back-ends. Figure 3.10 provides a simple illustration of the `ConcurrentHashMap` used by the Load Balancer.

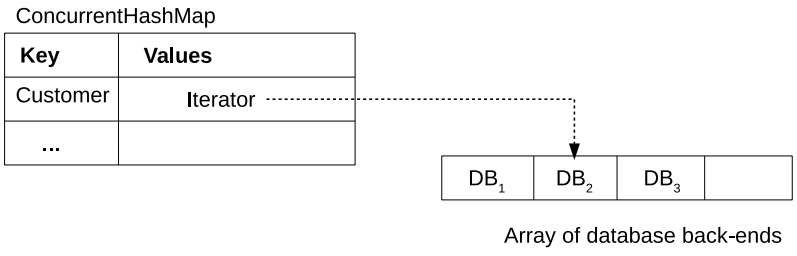

Figure 3.10: Data structures of the round-robin distribution implementation in the Sequoia controller

In the subsequent step, the controller performs additional processing of any constraints defined on the respective tables. Information about constraints is obtained from the Key Manager through calls to `GetPkPos()` and `GetFkPos()`. For each constraint type, there is unique logic that we describe below.

1. *Table contains primary key:* Before propagating the `INSERT` statement to the RAIDb-3 Load Balancer and to the appropriate database back-ends, the DBMS must update the Key Database in the Key Manager. As pointed out in the previous section, primary keys are generated internally by Sequoia and are unique among database back-ends. This is achieved via auto-generated primary key values in the Key Database.

   Consequently, when an `INSERT` statement for a table containing a primary key arrives, the Key Manager prepares the following statement for execution - Listing 3.21.

   Listing 3.21: Key Manager `INSERT` statement for primary key

   ```
   INSERT INTO name_of_the_tablePRM VALUES (DEFAULT);
   ```

   Here, `name_of_the_table` refers to the name of the table extracted from the original `INSERT` statement. The suffix `PRM` is appended to this name so that the statement can be executed on the previously created primary key table. In place of actual values, we are using the placeholder `DEFAULT`. Again, this is the HSQLDB specific syntax [13] which, when coupled with appropriate syntax in the `CREATE TABLE` statement, forces the DBMS to produce unique column-wide values for this record.

   In order to execute Statement 3.21 we use the JDBC function `executeUpdate()`

[18], with the additional argument `Statement.RETURN_GENERATED_KEYS`:

```
SqlStatement.executeUpdate("INSERT INTO "+aRequest.
    getTableName()+gPrmSuffix+" VALUES (DEFAULT);",
    Statement.RETURN_GENERATED_KEYS);
```

This particular argument permits fetching auto-generated values on successful completion of statement execution. In this way, we obtain a primary key value that is unique across database back-ends. However, this value needs to be embedded into the original `INSERT` request which will then be propagated to the RAIDB-3 Load Balancer and eventually to the appropriate database back-end. We accomplish this by post-processing of the original INSERT statement string.

It is important to note that in order to allow the Key Manager to successfully process the `INSERT` statement for tables containing the primary key, the value of the primary key in the original statement must not be fixed but instead represented by the placeholder `DEFAULT`. If this rule is not followed, the transaction fails. Of course, this `DEFAULT` placeholder is replaced by the generated value before being passed to the RAIDb-3 Load Balancer.

2. *Table contains foreign key:* In contrast to primary keys, foreign keys are not generated by Sequoia's RAIDb-3 system but are instead extracted from the original `INSERT` statement and subsequently inserted into the foreign key table of the Key Database. In order to extract the value of the foreign key from the original `INSERT` statement, we utilize information about the foreign key column index in the Key Manager, as well as additional string processing on the original `INSERT` statement.

On successful foreign key value extraction, the Key Manager prepares the statement shown in Listing 3.22 to be executed on the Key Database:

Listing 3.22: Key Manager `INSERT` statement for foreign key

```
INSERT INTO name_of_the_tableFRG VALUES (value);
```

Here, `name_of_the_table` is the name of the table extracted from the original `INSERT` statement. The suffix `FRG` is appended to this name so that the statement can be executed on the previously created foreign key table. Furthermore, "value" in the clause `VALUE` is only shown to better illustrate the basic principle. In practice, a *prepared statement* is used instead. No return value is collected for this statement. Note that the foreign key in the original statement is not modified in any way.

In order to demonstrate the core idea we will use the same table `INVOICE` from the previous example. We append a record to this table in the SQL client as shown in Listing 3.23.

Listing 3.23: Inserting record into table `INVOICE`

```
INSERT INTO Invoice VALUES(DEFAULT, 8, 91.89, '2016-02-04␣
    09:32:58');
```

The `INSERT` statement forces the Key Manager to execute two statements on the Key Database. Examples of the these statements are depicted in Listing 3.24 and Listing 3.25.

For the primary key:

Listing 3.24: Inserting primary key in Key Manager

```
INSERT INTO InvoicePRM VALUES (DEFAULT);
```

and for the foreign key:

Listing 3.25: Inserting foreign key in Key Manager

```
INSERT INTO InvoiceFRG VALUES (8);
```

Furthermore, the Key Manager modifies the original statement so as to explicitly indicate the value of the primary key - Listing 3.26.

Listing 3.26: Modified original `INSERT` statement

```
INSERT INTO Invoice VALUES(10, 0, 91.89, '2016-02-04␣
    09:32:58');
```

Here the value "10" implies that the primary key was obtained in the course of execution of Statement 3.24 and that the DBMS hosting the Key Database generated this value.

The basic workflow for the `INSERT` statement is depicted in Figure 3.11. From an architectural perspective, the (simplified) process of handling the `INSERT` query in RAIDb-3 is illustrated by Figure 3.12. In Figure 3.12, numbers in '()' denote the process sequence number.

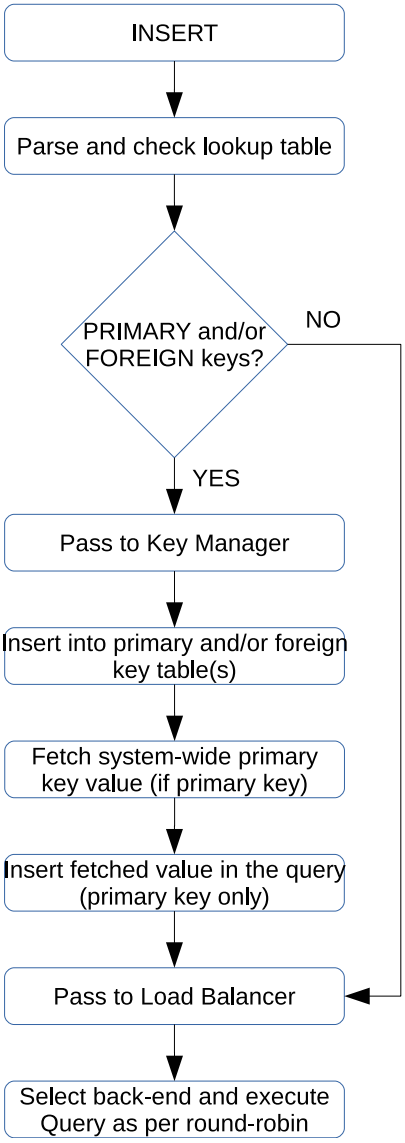

Figure 3.11: `INSERT` statement workflow.

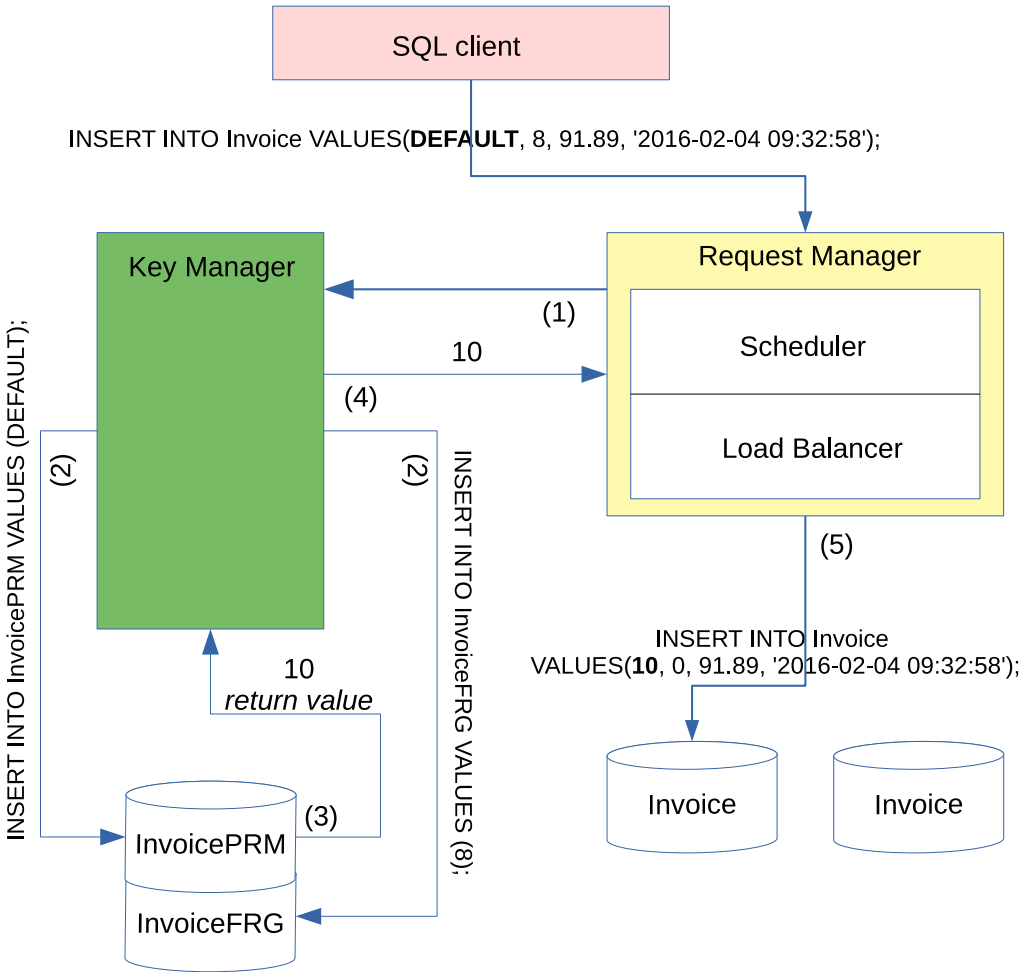

Figure 3.12: `INSERT` statement processing overview.

3.7.5 Handling the `SELECT` statement

Implementing `SELECT` functionality is the most complex part of RAIDb-3. Besides adding new features to Sequoia, it was necessary to modify several fundamental components of the system. Considering the RAIDb-3 objectives, the operation of retrieving data from database back-ends can and should be implemented in a concurrent manner. We therefore implemented the respective functionality within the RAIDb-3 Load Balancer component.

We note that the original Sequoia DBMS contains basic data management primitives such as task structures and a task queue that are based on a thread pool construct. These primitives allow us to build a concurrent RAIDb-3 data fetching mechanism in which queries are executed asynchronously by placing the respective query task into a task queue, thereby allowing the partial results to be joined together within a thread-safe result container.

3.7.5.1 RAIDb-3 Load Balancer

The RAIDb-3 Load Balancer implementation utilizes a task class that extends the original Sequoia structure. This class houses the query to be executed on the database back-ends. The Load Balancer uses a round-robin algorithm to ensure that data is distributed evenly among database back-end nodes. In order to retrieve data from all database back-ends, the concurrent RAIDb-3 Load Balancer issues common queries to all relevant back-ends. In short, the Load Balancer generates as many query tasks as there are active database beck-ends.

During this process, a query task object is added to the task queue immediately

after its creation. The function that adds a task to the task queue returns immediately, without waiting until execution of the embedded query completes. As soon as the task object is placed into the task queue, the supporting thread-pool begins execution of the task. Therefore, if we have a RAIDb-3 cluster consisting of three database back-ends, and the master node that receives and processes the user query runs on appropriate hardware and operating system (e.g., a computer with three cores or three CPUs), retrieval from all database back-ends can be accomplished in parallel.

Note that the task object contains an internal data member that maintains intermediate results obtained during execution of a query on a given database back-end. However, once execution completes this intermediate result needs to be merged into the data structure representing the full result from all database back-ends. Tasks executing on distinct back-ends may of course complete their execution at different times. That said, there is a possibility that at least two tasks might complete execution at the same time. Therefore, while designing the result data structure we include locking mechanisms to preventing data corruption due to simultaneous writing. Finally, besides aggregating partial results, our aggregation structure – `RaControllerResultSet` - supports the joining of results for two tables. We elaborate on the functionality of the result data structure later in this chapter.

Prior to its propagation to the RAIDb-3 Load Balancer, the original query undergoes significant processing. In fact, the functionality involved in this processing greatly depends on the type of query – whether this is a non-join or join query.

3.7.5.2 Non-join `SELECT` statement

Handling of non-join `SELECT` statements is relatively straightforward. The Request Manager receives the original SELECT query from the SQL client and then passes

it to the Key Manager. In addition, the Request Manager processes the statement string and evaluates the length of the table list. For non-join statements, the table list should not be larger than one. If the Key Manager determines that the query is a non-join, it performs no further processing but appends it to the empty result array and returns it back to the Request Manager.

The Request Manager iterates through the received array of `SELECT` queries – which is of size one for a non-join query – and propagates each query down to the RAIDb-3 Load Balancer. This latter component concurrently executes the queries on data back-ends. The final result is sent to the JDBC driver which, in turn, returns it to the SQL client.

3.7.5.3 Join `SELECT` statement

Similar to the previous case, the Request Manager passes the `SELECT` statement to the Key Manager, which examines the table list of this statement. A join `SELECT` statement is performed on two tables. Consequently, the Key Manager begins the additional processing associated with handling join queries.

Specifically, it initializes a lookup table in which it binds the name of that table with the constraint type. As we mentioned earlier, we only consider joins between primary and foreign keys. As such, the Key Manager determines what type of constraint – primary key or foreign key - corresponds to each table in the table list. To do this, the Key Manager analyses the original statement string. We consider, for example, two tables created as per Listing 3.27 and Listing 3.28.:

Listing 3.27: Creation of table `CUSTOMER`

```
CREATE TABLE Customer(CustomerId INTEGER GENERATED ALWAYS AS
    IDENTITY PRIMARY KEY, CustomerName VARCHAR(64),
    CustomerAge INTEGER, CustomerSex CHAR(1),
    CustomerProvince CHAR(2), CustomerLocation VARCHAR(64));
```

Listing 3.28: Creation of table `INVOICE`

```
CREATE TABLE Invoice(InvoiceId INTEGER GENERATED ALWAYS AS
    IDENTITY PRIMARY KEY, InvoiceCustomerId INTEGER FOREIGN
    KEY REFERENCES Customer(CustomerId), InvoiceTotal DOUBLE,
    InvoiceTimeStamp TIMESTAMP);
```

Then, we issue the JOIN `SELECT` query as in Listing 3.29.

Given this scenario, the Key Manager would create the lookup table shown in Table 3.24.

On the subsequent step, the Key Manager generates the JOIN statement to be executed on the Key Database. The Key Manager does this is the following manner.

1. It extracts the names of the tables involved in the JOIN operation from the original query.

Listing 3.29: Arbitrary join `SELECT` query

```
SELECT * FROM Invoice, Customer WHERE
    Invoice.InvoiceCustomerId=Customer.CustomerId AND
    Invoice.InvoiceTotal>=50 AND
    Invoice.InvoiceTimeStamp>='2016-02-01 00:00:00' AND
    Invoice.InvoiceTimeStamp<='2016-02-28 23:59:59' AND
    Customer.CustomerAge>=30 AND Customer.CustomerAge<40 AND
    Customer.CustomerSex='M' AND
    Customer.CustomerProvince='QC';
```

Table 3.24: Key Manager constraint map

Table	Constraint used in join
Invoice	Foreign key
Customer	Primary key

Thus, if our running example is considered, the results for Listing 3.29 are: `Invoice` and `Customer`.

2. It consults the lookup table in order to determine the constraint type that corresponds to the relevant table and appends the appropriate suffix to the table name. A suffixed table name corresponds to one of the tables in the Key Manager database.

   For our example the information about constraint type is acquired from the lookup table depicted in Table 3.24. The following table name transformations are performed:

   `Customer` → `CustomerPRM`

   `Invoice` → `InvoiceFRG`

3. Using information about the constraint types, the Key Manager builds a JOIN predicate by equating the relevant primary key column with a foreign key column.

   For our example the Key Manager determines primary/foreign key columns as follows:

   `Customer` → `CustomerPRM` → `PRIMARY_KEY`

   `Invoice` → `InvoiceFRG` → `FOREIGN_KEY`

Finally, the query predicate assumes the following form:

`InvoiceFRG.FOREIGN_KEY=CustomerPRM.PRIMARY_KEY`

Thus, the general query form produced by the Key Manager can be expressed as per Listing 3.30.

Listing 3.30: Key Manager join query on key tables

```
SELECT * FROM table_1PRM INNER JOIN table_2FRG ON
    table_1PRM.PRIMARY_KEY=table_2FRG.FOREIGN_KEY;
```

We use an *INNER JOIN* so that the results of the query contain only records that have matching values in both tables. The results obtained on execution of this statement include only those primary keys of `table_1` and only those foreign keys of `table_2` that correspond to the result expected on execution of the original join `SELECT` query on Sequoia's virtual database.

For our previous example the join statement on the Key Database takes the form as depicted in Listing 3.31.

Listing 3.31: Key Manager JOIN query on tables `InvoiceFRG` and `CustomerPRM`

```
SELECT * FROM InvoiceFRG INNER JOIN CustomerPRM ON
    InvoiceFRG.FOREIGN_KEY=CustomerPRM.PRIMARY_KEY;
```

Execution of Statement 3.30 produces a table of two integer columns. Specifically, one column corresponds to the primary key of one source table joined on its primary key – we call it the *primary key set*. The second column corresponds to the foreign key of the second source table joined on its foreign key – we call it the *foreign key set*.

As soon as the result from the Key Database arrives, the Key Manager prepares

two non-join SELECT queries, which include an IN predicate on the values corresponding to the primary or foreign key set, one for each table from the original SELECT query. A generic query format can be specified as in Listing 3.32 and Listing 3.33.

Listing 3.32: Querying primary keys

```
SELECT * FROM table_1 WHERE table_1.primary_key IN (
    (UNNEST(?));
```

Here, the IN predicate contains the values of the primary key set.

Listing 3.33: Querying foreign keys

```
SELECT * FROM table_2 WHERE table_2.foreign_key IN (
    (UNNEST(?));
```

In this case, the IN predicate contains the values of the foreign key set.

When generating these queries we use HSQLDB-specific syntax for the predicate IN [13] - UNNEST allows us to use an array for the IN predicate value. This syntax is intended to be used with *prepared statements* where a variable length array of values of type java.sql.Array [18] can be used as the parameter. Additionally, the Key Manager performs filtering of the original SELECT query in order to remove primary and foreign key predicates from the WHERE component, as well as keeping only those predicates that are relevant to the respective table. Thus, if we consider the original join SELECT example in 3.29, the two corresponding non-join queries generated by the Key Manager take the form given in Listing 3.34 and Listing 3.35.

Listing 3.34: Database back-end query for table INVOICE

```
SELECT * FROM invoice WHERE invoice.invoicecustomerid IN (
    (UNNEST(?)) ) AND Invoice.InvoiceTotal>=50 AND
    Invoice.InvoiceTimeStamp>='2016-02-01 00:00:00' AND
    Invoice.InvoiceTimeStamp<='2016-02-28 23:59:59';
```

Listing 3.35: Database back-end query for table `CUSTOMER`

```
SELECT * FROM customer WHERE customer.customerid IN (
    (UNNEST(?)) ) AND Customer.CustomerAge>=30 AND
    Customer.CustomerAge<40 AND Customer.CustomerSex='M' AND
    Customer.CustomerProvince='QC'';
```

We note that in order to support our approach, we had to modify the original Sequoia `SelectRequest` class that represents the `SELECT` query so as to allow for the capture of the array associated with the `IN` predicate. Ultimately, each generated non-join query represented by the `SelectRequest` class is saved in a result array alongside the respective array of primary or foreign keys from the corresponding primary or foreign sets.

The Request Manager receives the array of `SELECT` queries for the individual tables, iterates through this array – which is of size two for a join query – and delivers each query to the RAIDb-3 Load Balancer that concurrently executes them on the data back-ends. Intermediate results for each query from the array are saved in a `RaControllerResultSet` construct. The final result is produced by joining together all intermediary results in a manner similar to join `SELECT`. The final query result is sent to the JDBC driver which then relays it to the SQL client.

The workflow logic for the `SELECT` statement is depicted in Figure 3.13.

We can generalize the architectural approach discussed in this section with the illustration in Figure 3.14

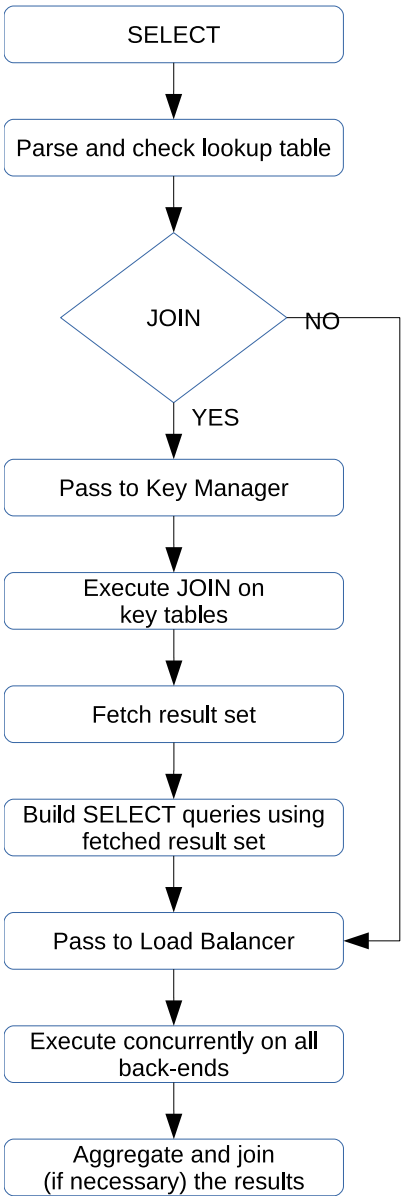

Figure 3.13: The `SELECT` statement algorithm.

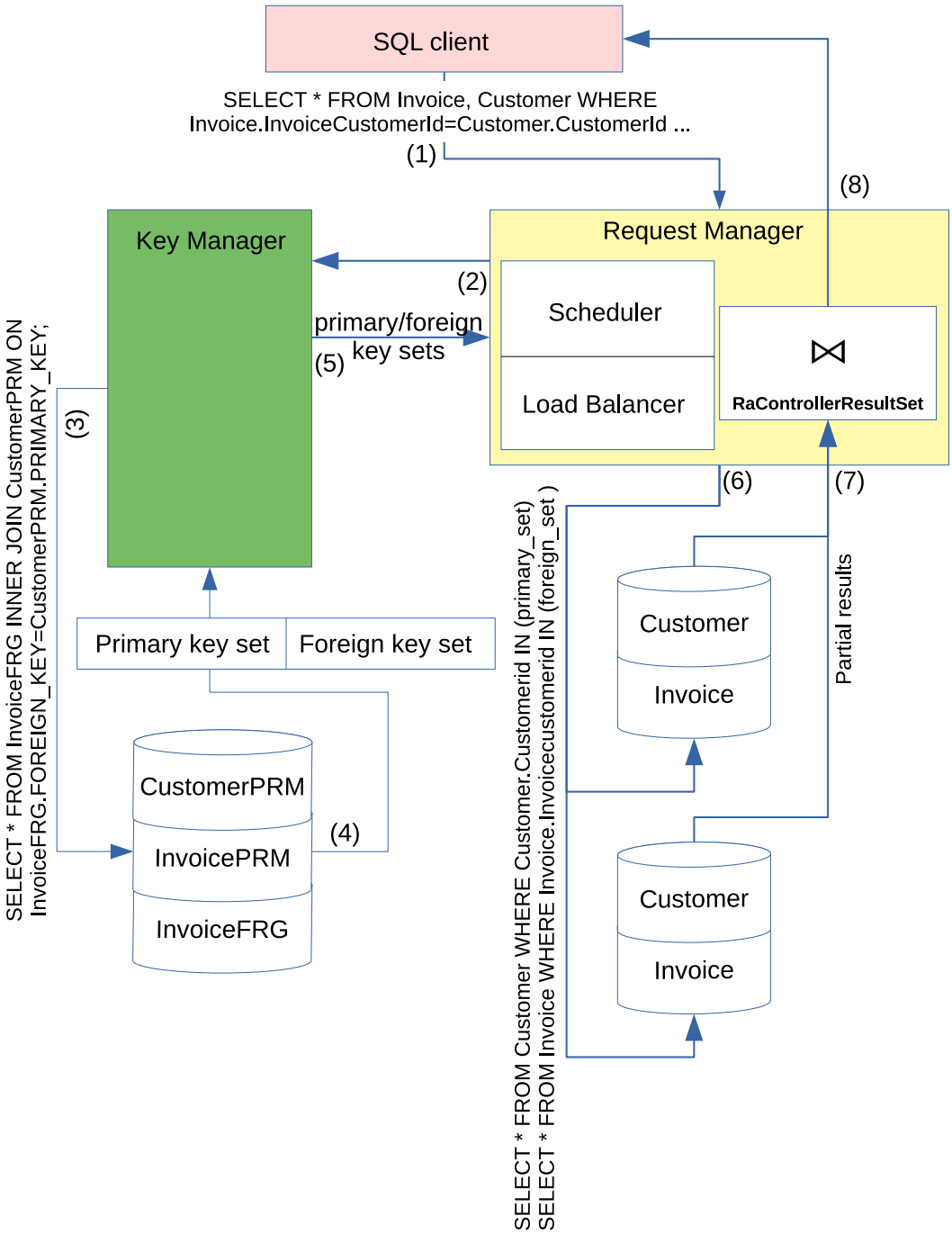

Figure 3.14: The `SELECT` statement processing overview.

3.7.5.4 `RaControllerResultSet` – RAIDb-3 result data structure

In the course of the discussion above, we have mentioned a specially designed data structure called a `RaControllerResultSet` that is used to store, aggregate, and join intermediate results, as well as generating the final query result. Below, we describe the features of this data structure in detail.

The `RaControllerResultSet` class stores a representation of the result obtained during execution of the `SELECT` statement. This includes result metadata such as the data type of the result columns, as well as the actual data. Metadata is saved within an array of fields [18]: `ArrayList<Field>`.

while the actual data is saved in a `HashMap` [18]:

```
HashMap<Integer , ArrayList<ArrayList<Object>>>
```

Here, the key is represented by the value of the constraint – a primary or foreign key depending on the structure of the join query. We introduce a new term - *"match column"* - which denotes the column containing the constraint on which the original `SELECT` query is performed .

For example, if we consider query 3.34, we know that the constraint used in the original SELECT statement 3.29 for the `INVOICE` table is `Invoice.Invoicecustomerid`. Therefore, values of column `Invoice.Invoicecustomerid` are used as keys in the `HashMap` and the column `Invoice.Invoicecustomerid` is a match column.

The value in the `HashMap` is represented by an array of rows which, in turn, is also an array of Objects [18]. This is the case because of the fact that for a given key in the `HashMap`, its value may contain more than one row.

In addition, the `RaControllerResultSet` contains several additional data members that are responsible for synchronizing access to `RaControllerResultSet` internals, as well as storing internal states and data.

We designed the `RaControllerResultSet` with the intent to accomplish a number of tasks, including:

1. Aggregating intermediate result data from the database back-ends. As we stated above, we designed the Load Balancer so that SELECT queries are issued on all active database back-ends and execution is performed concurrently. Intermediate results are then aggregated into one result set. The last step is performed by the `RaControllerResultSet Append()` method.

   The operation itself is rather straightforward. Simply put, metadata is initialized with data contained in the argument of the `Append()` on the first invocation of this method. This is the case because aggregation is performed on the results of a query issued for the same table metadata. The array housing the actual result data is also extended with rows obtained from the argument passed to this method. In order to avoid data corruption, both operations are protected by Java synchronization mechanisms.

   The functionality of the `Append()` method of the `RaControllerResultSet` is illustrated in Figure 3.15. Note that `CustomerId` is a match column.

2. Joining result data of two tables. When the Request Manager processes a JOIN `SELECT` query, it creates two non-join queries to be executed on all active database back-ends. In the course of execution of these queries, intermediate results of execution for each respective table are aggregated (as described above)

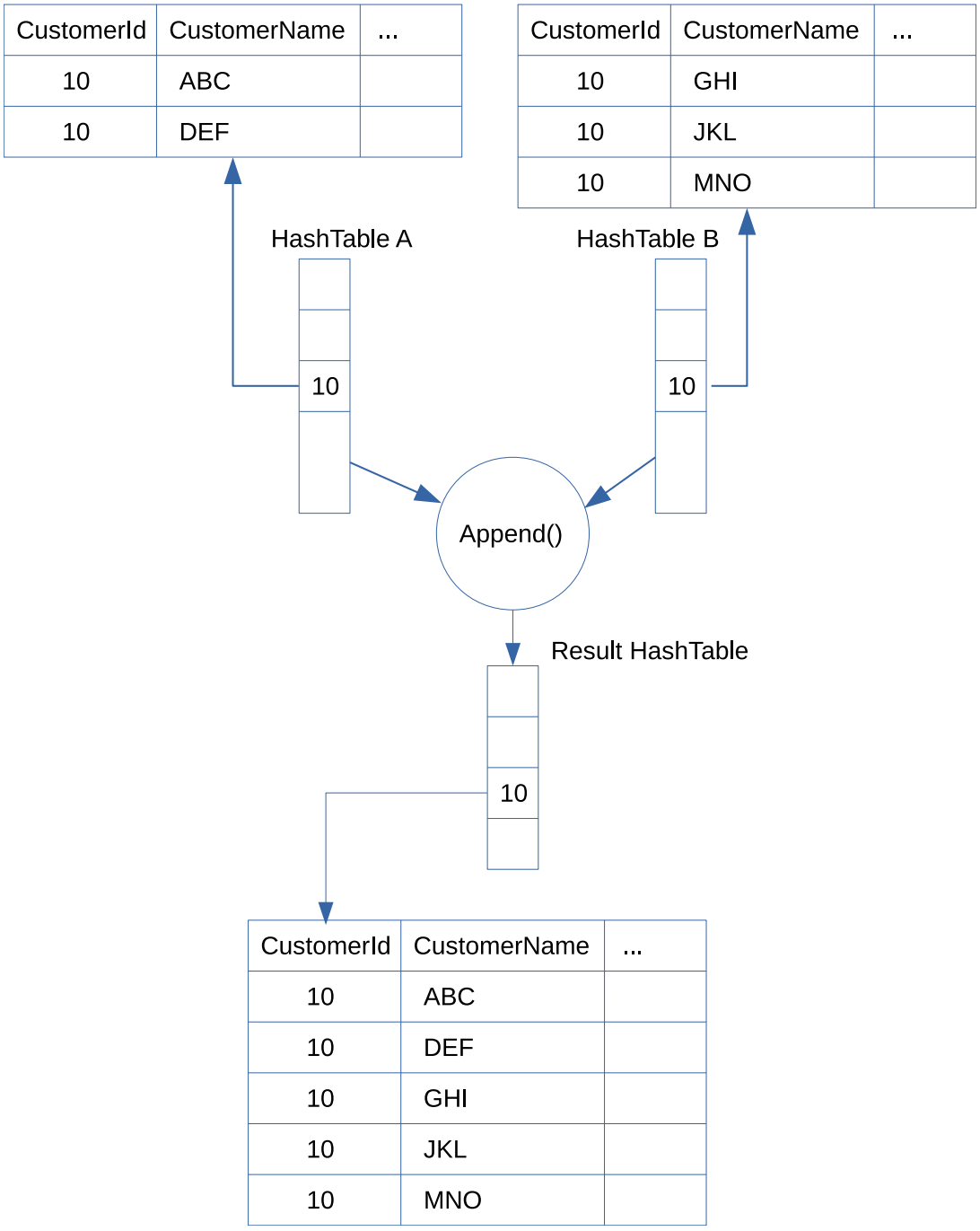

Figure 3.15: The `Append()` method.

into two set of results. These two sets need to be joined into a final single result. This function is carried out by the `RaControllerResultSet Join()` method.

Joining is implemented as follows. In its initial state, the `RaControllerResultSet` contains a fully aggregated result for one of the tables from the original `SELECT` query (i.e., the result obtained by execution of Query 3.32). Just before calling `Join()`, the Request Manager receives the fully aggregated result for another table from the original `SELECT` query (i.e., the result of execution of Query 3.33). The Key Manager uses the latter as an argument in the call to `Join()`. The `RaControllerResultSet` first merges the metadata of the two result sets. To do so, it simply extends the existing ArrayList [18] with fields from the argument metadata. The process of joining the metadata is illustrated in Figure 3.16.

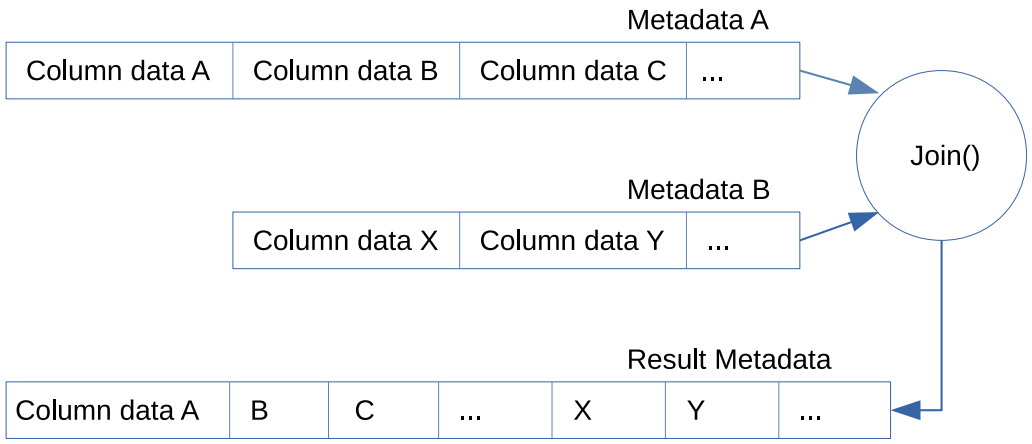

Figure 3.16: Application of the method `Join()` to metadata.

Finally, in order to join the result data, the `RaControllerResultSet` iterates through the values of the existing data members in the `HashMap`. On each iteration, the key value is extracted and used to fetch its matching value from the argument

data member. Fetching is performed with `O(1)` time complexity. If the argument data member exists for a given key, the `RaControllerResultSet` performs a merge of two arrays of rows – one array is from the existing `RaControllerResultSet` and the second is from the argument passed to `Join()`. Time complexity of this operation is `O(`$n \times k$`)`, where n and k are the sizes of the arrays. When the `RaControllerResultSet` finishes iterating through the internal data member, it will contain the final result of joining the two tables. Figure 3.17 illustrates this concept.

In the initial version of the `RaControllerResultSet`, we experimented with two constructs provided within Java and intended for executing off-line join `SELECT` queries – the `CachedRowSet` and `JoinRowSet` [18]. However, our tests showed unacceptable performance by these constructs, even for moderate sizes of input data (1000-10000 records). Unfortunately, we were unable to further investigate these two classes as their source code is not provided within the Java API. Furthermore, according to the Java API documentation [18], they are subject to possible deprecation in future versions of the JDK. Consequently, we chose to implement our own mechanism to join the result sets.

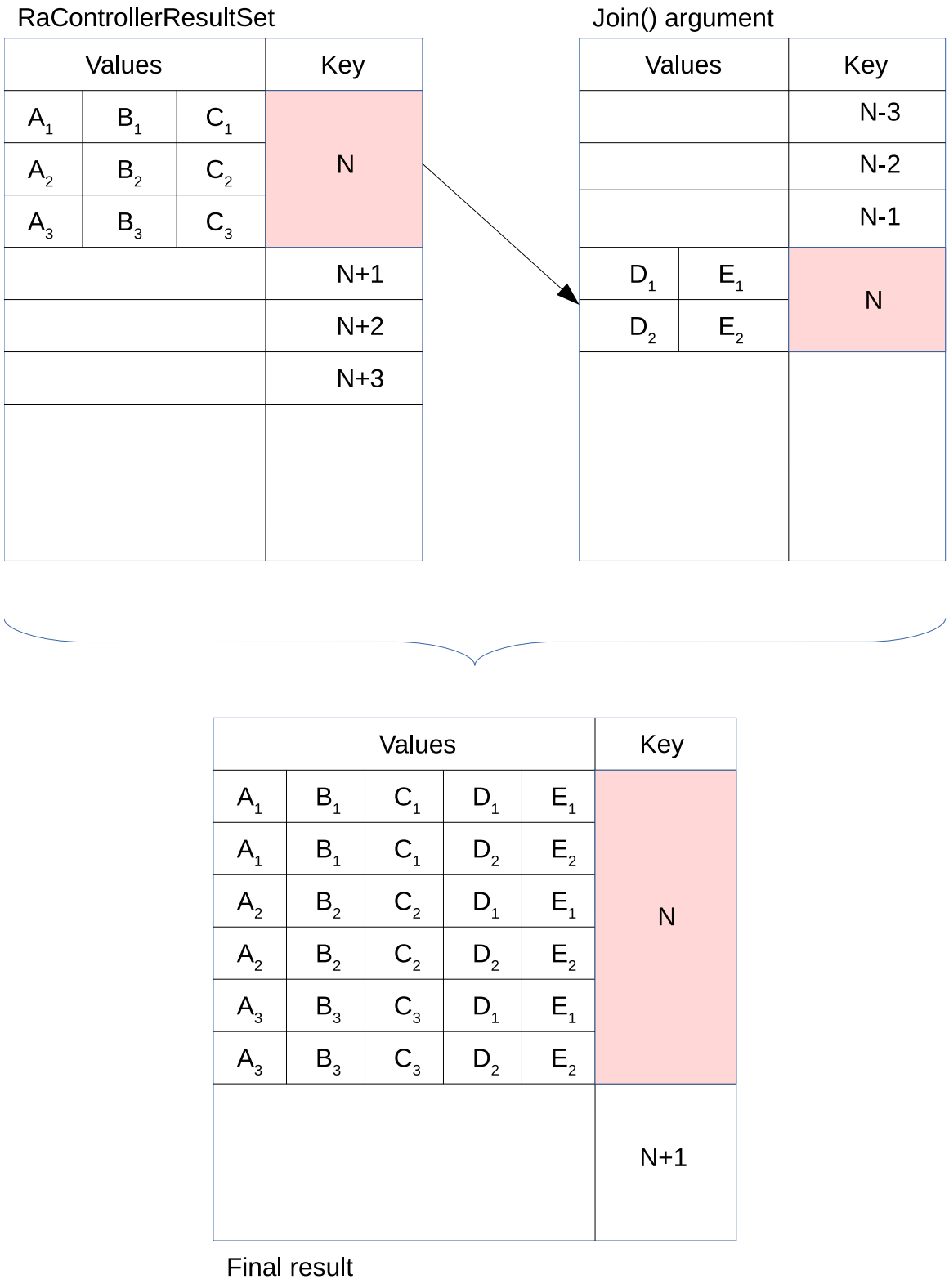

Figure 3.17: Illustration of the `RaControllerResultSet Join()`

3.7.6 Additional extensions

In the previous sections, we have covered the principle implementation steps relevant to the RAIDb-3 design. However, we note that the complete implementation was considerably broader and includes modifications or new developments in many other components of Sequoia. This include (but is not limited to):

- Modifying the management of virtual database schemas.
- Adapting the Sequoia driver interface so as to provide for integration of the `RaControllerResultSet`.
- Modifying classes responsible for representing SQL statements: `CREATE TABLE`, `INSERT`, `SELECT`.
- Extending the Task Manager with the ability to process `SELECT` queries concurrently.

3.8 Conclusion

In this chapter we have provided a detailed overview of the existing Sequoia infrastructure, as well as an in-depth description of the implementation of the RAIDb-3 model in Sequoia. Specifically, we elaborated on the following topics:

- Motivation and concept of the RAIDb-3 model.
- Design considerations for RAIDb-3.
- Details of the implementation of the key components of the model.

In the next chapter we will present a series of experimental results that illustrate the capability of the core design, as well as limitations in the current prototype.

Chapter 4

Evaluation

4.1 Introduction

In this research we set an objective to design a distributed DBMS built on the principles of table-row granularity among the supporting database back-ends. We called this model RAIDb-3. In previous chapters, we thoroughly examined the core design principles as well as the implementation specifics of RAIDb-3. However, to properly ground the research we need to answer questions about how well the model performs in comparison to standard DBMSes in terms of common factors that affect performance.

In our evaluation, we in fact distinguish between two performance aspects: *qualitative* and *quantitative*. Below, we elaborate on both of these issues.

4.2 Preliminaries and test environment

Given the focus on distributed databases, our performance evaluation requires reasonably large data set populations. In order to generate such sets we developed a C++ application (SqlGen) that generates script files containing both SQL statements and random data (we will cover the structure of this data in later sections). The size and

the structure of the generated data can be adjusted through parameters described in Table 4.1.

Table 4.1: SqlGen command line parameters

Parameter	Meaning
–customernum	Number of rows in table `CUSTOMER`
–invoicenum	Number of rows in table `INVOICE`. This table is created to support joins with the `CUSTOMER` table. The parameter can have a zero value; in this case, no `INVOICE` table is generated by the script.
–ratio	Number in interval (0; 1] that defines the maximum size of a subset of uniformly-distributed primary keys from the `CUSTOMER` table (i.e. $subset\_size = number\_of\_primery\_keys \times ratio$) which can be referenced in table `INVOICE`.

In addition, we use SQLTool, a JDBC compatible console client to connect to Sequoia and to execute statements from the script file. SQLTool provides several convenient features such as measuring the time of execution and saving query results into external files. SQLTool is shipped alongside HSQLDB [13].

Tests were performed on conventional hardware. In this case, we used a machine equipped with an AMD Ryzen CPU [4] with eight physical cores (architecturally allowing us to run two simultaneous threads on each core) and 32 Gb RAM running under the Fedora 29 [10] Linux operating system. We acknowledge that this environment does not necessarily represent enterprise level data management systems equipped with considerably more powerful hardware and, as such, might not precisely reflect results for extremely large data populations of 10-s and 100-s of millions of

records. That said, it is quite capable of adequately evaluating data sets and query forms relevant to the objectives of this research.

As a final point, we note that in this work we did not evaluate more complex use cases such as: (1) running a whole system or its parts on virtual machine(s), or (2) exploiting distribution across a network (i.e., where one or several database back-ends are located on remote network nodes). In terms of the later, we note that this does negatively impact the current test results as communication costs can not be amortized across the network fabric.

4.3 Qualitative and quantitative performance evaluation

Because the model proposed in this research is novel in its design, there is no existing test set that can be directly used to assure the validity of the RAIDb-3 cluster. As such, it is imperative that we be able to assess the quality and correctness of the results returned by the Sequoia RAIDb-3 cluster.

Fortunately, ensuring the validity of the data, in the current context, at least, is rather straightforward and can be accomplished by simply comparing the results returned by the RAIDb-3 cluster with the results returned by a reference standalone DBMS (e.g., HSQLDB) for the same data set. We verify correctness for all queries used in our evaluation.

For the case of quantitative evaluation, we examine how well the RAIDb-3 model performs when compared with a conventional DBMS such as HSQLDB. Presently, there are a few well-established techniques designed to evaluate DBMS performance. A brief review of these techniques is listed in [28]. In most cases, each evaluation

mechanism proposes certain performance metrics and establishes benchmarks that are then used to evaluate various aspects of the DBMS.

One of the most universal evaluation solutions is offered by TPC – Transaction Processing Performance Council [30]. TPC has developed a suite of benchmarks for transaction processing and database performance analysis. Each individual benchmark targets specific application-dependant areas such as virtualisation, big data, decision support, etc. From the wide range of benchmarks, TPC-H [31] can be considered as the most relevant to our research.

TPC-H is a decision support benchmark that consists of a suite of business oriented ad-hoc queries. Data and queries were selected so as to have broad industry-wide relevance. The TPC-H benchmark operates on large volumes of data (millions of records) and executes queries with a high degree of complexity. TPC-H uses the TPC-H Composite Query-per-Hour Performance Metric (QphH@Size) to evaluate performance.

As we mentioned above, TPC-H is a suite of queries. Unfortunately, some of the queries comprising TPC-H may not be used with the current RAIDb-3 model due to the limitations in the prototype implementation. For example, many of the TPC-H queries are join `SELECT` queries that involve more than two tables and/or have `WHERE` clauses containing numerous conditions that are not necessarily established between primary and foreign keys. These factors prevent us from using the full TPC-H benchmark directly. However, we developed a reduced version of the TPC-H benchmark that allows us to evaluate the core elements of the RAIDb-3 model.

Below are the two primary features of our evaluation technique, as derived from the TPC-H framework:

- Usage of a single SELECT query in two scenarios – non-join and join queries.
- Measure of time consumed by execution of query (T_e) as a metric of performance (instead of the more complex QphH@Size).

In our evaluation we strive to follow the general TPC-H approach of designing relatively complex queries. We evaluate performance for various data population sizes. Additionally, we examine performance for different RAIDb-3 cluster configurations (i.e., for different database back-end counts – 3, 6 and 13).

Ultimately, in order to measure query time we follow the approach proposed in [12]. To be precise, we run every query ten times and measure elapsed time. Final results are averaged. We do not need to use external methods of measuring query time, as HSQLDB's SqlTool offers this functionality. A more advanced technique is proposed in [26], and takes into consideration a wide range of factors such as I/O and network time. However, due to the complexity of this measurement technique, we do not employ that approach at the current time.

In terms of the RAIDb-3 model, there are two key test cases - non-JOIN and JOIN query; therefore, we will consider a number of scenarios that include:

- Non-JOIN `SELECT` queries on arbitrary data sets for cases of 3, 6 and 13 database back-ends in a RAIDb-3 cluster;
- JOIN `SELECT` queries on arbitrary data sets for cases of 3, 6 and 13 database back-ends in a RAIDb-3 cluster.

4.3.1 Non-JOIN queries

For non-JOIN `SELECT` queries we use a table with the structure described in Table 4.2 (in general, a table can have arbitrary structure).

Table 4.2: Structure of table `CUSTOMER`

Column name	Column type	Constraint type
CustomerId	INTEGER	Primary key
CustomerName	VARCHAR(64)	
CustomerAge	INTEGER	
CustomerSex	CHAR(1)	
CustomerProvince	CHAR(2)	
CustomerLocation	VARCHAR(64)	

Listing 4.1: Statement for creating table `CUSTOMER`

```
CREATE TABLE Customer(CustomerId INTEGER GENERATED ALWAYS AS
    IDENTITY PRIMARY KEY, CustomerName VARCHAR(64),
    CustomerAge INTEGER, CustomerSex CHAR(1),
    CustomerProvince CHAR(2), CustomerLocation VARCHAR(64));
```

The table is created by issuing the SQL statement shown in Listing 4.1.

This table is populated with random data generated by SqlGen. The script essentially contains a series of `INSERT` statements. Below, an example of one such statements is provided.

Listing 4.2: Sample of statement used for populating table `CUSTOMER`

```
INSERT INTO Customer VALUES(DEFAULT, 'Jami Byrne', 57, 'M',
    'QC', 'Grande-Riviere');
```

We examine three cases with data populations of 100,000 (we will refer tho this as *Population 1*), 1,000,000 (*Population 2*) and 10,000,000 (*Population 3*) records. Additionally, we devise three distinct `SELECT` queries so as to produce results of different sizes and, hence, to better substantiate validity of our findings. We refer to these queries as Query X, Query Y, and Query Z. The queries are shown in Listings 4.3, 4.4, and 4.5.

Listing 4.3: Query X

```
SELECT * FROM Customer WHERE CustomerProvince='NL' AND
    CustomerAge>37 AND CustomerAge<=39 AND CustomerSex='M'
    AND CustomerLocation='Mount Pearl';
```

Listing 4.4: Query Y

```
SELECT * FROM Customer WHERE CustomerAge<25 AND
    CustomerSex='M' AND CustomerLocation='Nanaimo';
```

Listing 4.5: Query Z

```
SELECT * FROM Customer WHERE CustomerProvince='QC';
```

In accordance with our methodology, we evaluate the validity of the results returned by the RAIDb-3 cluster for each query and population. Therefore, we compare results produced by all tested RAIDb-3 configurations (3, 6 and 13 database back-ends) with results returned by a standalone HSQLDB. Generalized validity check results are listed in Table 4.3.

Table 4.3: HSQLDB and Sequoia RAIDb-3 cluster result match

	RAIDb-3, 3 back-ends	RAIDb-3, 6 back-ends	RAIDb-3, 13 back-ends
Match with results from HSQLDB	✓	✓	✓

4.3.1.1 Population 1

For each query type we receive final result sets of the sizes listed in Table 4.4.

Table 4.4: Size of result sets for test queries

Query type	Result set size
X	35
Y	5
Z	7754

After performing a sequence of query executions we obtain quantitative results listed in the Appendix, in Table A.1 - Table A.3. Table 4.5 summarizes these results.

Table 4.5: Average non-JOIN query execution time for table `CUSTOMER` containing Population 1

Database system	Query X time, ms	Query Y time, ms	Query Z time, ms
HSQLDB	35	25	48
RAIDb-3, 3 back-ends	36	24	78
RAIDb-3, 6 back-ends	32	20	77
RAIDb-3, 13 back-ends	29	18	78

4.3.1.2 Population 2

Sizes of the result sets for all respective queries are listed in Table 4.6.

Table 4.6: Size of result sets for test queries

Query type	Result set size
X	304
Y	57
Z	76894

On performing a sequence of query executions we obtain quantitative results listed in the Appendix, in Table A.4 - Table A.6. Table 4.7 summarizes these results.

Table 4.7: Average non-JOIN query execution time for table `CUSTOMER` containing Population 2

Database system	Query X time, ms	Query Y time, ms	Query Z time, ms
HSQLDB	208	152	266
RAIDb-3, 3 back-ends	118	86	430
RAIDb-3, 6 back-ends	91	61	435
RAIDb-3, 13 back-ends	70	46	473

4.3.1.3 Population 3

Table 4.8 lists sizes of result sets for all respective queries.

Table 4.8: Size of result sets for test queries

Query type	Result set size
X	3150
Y	611
Z	769776

After performing a sequence of query executions we obtain quantitative results listed in the Appendix, in Table A.7 - Table A.9. Table 4.9 summarizes these results.

Table 4.9: Average non-JOIN query execution time for table `CUSTOMER` containing Population 3

Database system	Query X time, ms	Query Y time, ms	Query Z time, ms
HSQLDB	1823	1647	2108
RAIDb-3, 3 back-ends	851	693	5929
RAIDb-3, 6 back-ends	535	441	5828
RAIDb-3, 13 back-ends	391	309	6366

4.3.1.4 Discussion

The results obtained in terms of data correctness are perfectly consistent across all three query tests. Furthermore, we note that this type of analysis was repeated multiple times throughout the research program.

In the course of the quantitative evaluation we discovered that query execution performance of the RAIDb-3 controller is significantly affected by the structure of the query. Below, in Figures 4.1 - 4.3, we provide diagrams that summarize our findings.

The results clearly indicate the performance advantage of the RAIDb-3 controller over a standalone HSQLDB for Query X and Query Y (i.e., the queries that return relatively small result sets). In particular, we can observe a trend that relative RAIDb-3 query execution performance increases with population size growth (e.g. a RAIDb-3 configuration with 13 back-ends on execution of Query Y gives a 3.3 times advantage over HSQLDB for Population 2 while the same query delivers a 5.33 times advantage for larger Population 3). In fact, this effect was expected as the horizontal partitioning design implemented in the RAIDb-3 controller provides for increased parallelism and, thus, allows the system to deliver better query execution times relative to a standalone DBMS.

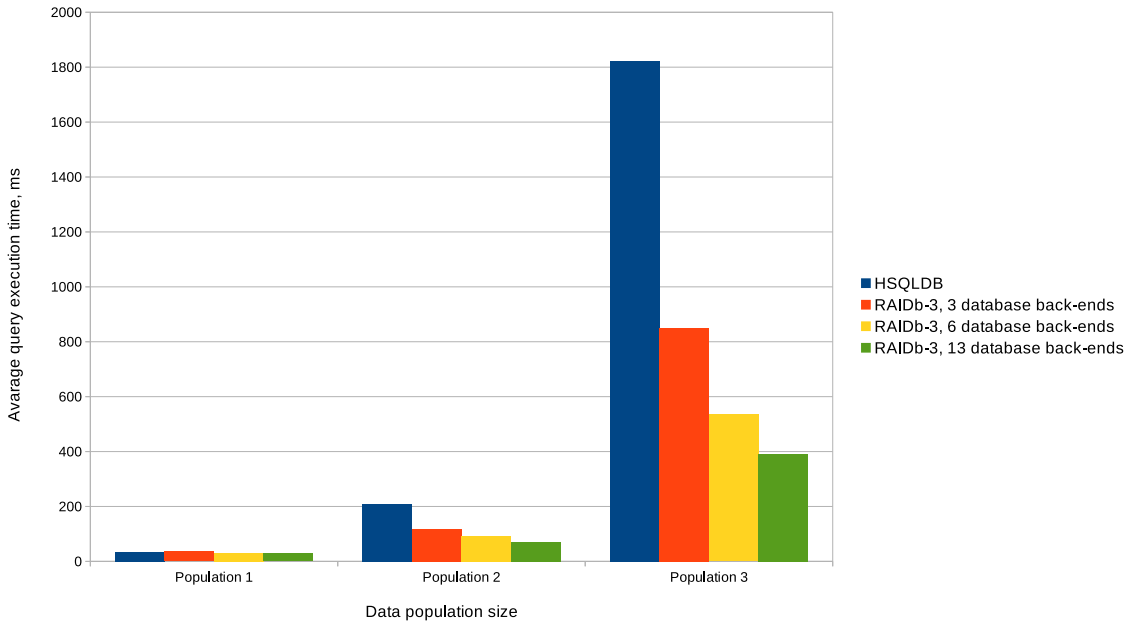

Figure 4.1: Summarized results for non-JOIN Query X

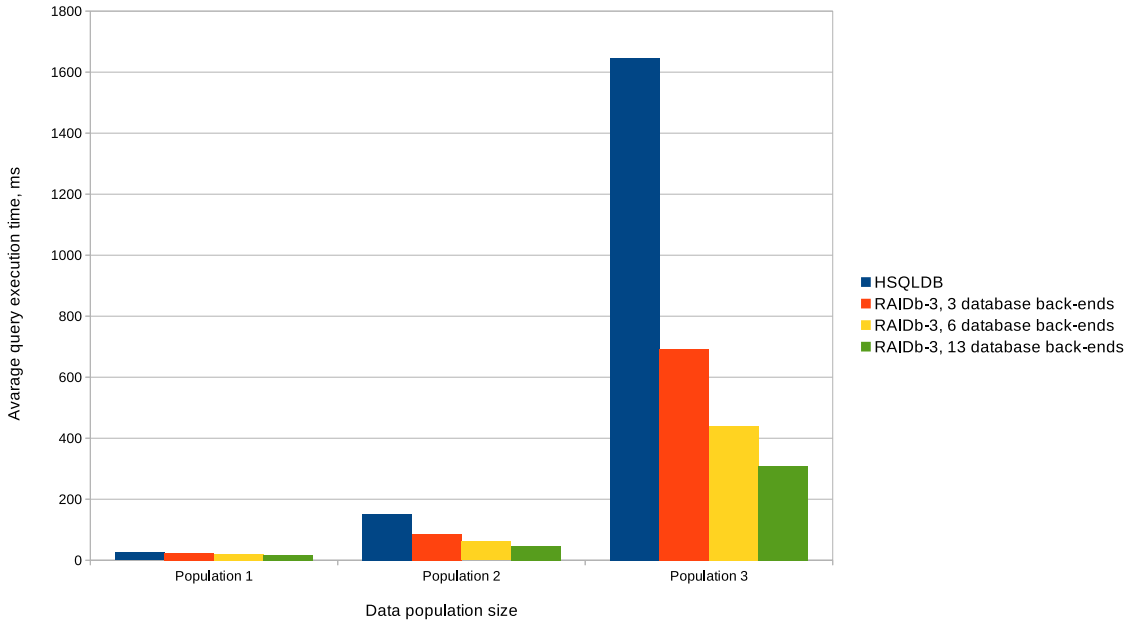

Figure 4.2: Summarized results for non-JOIN Query Y

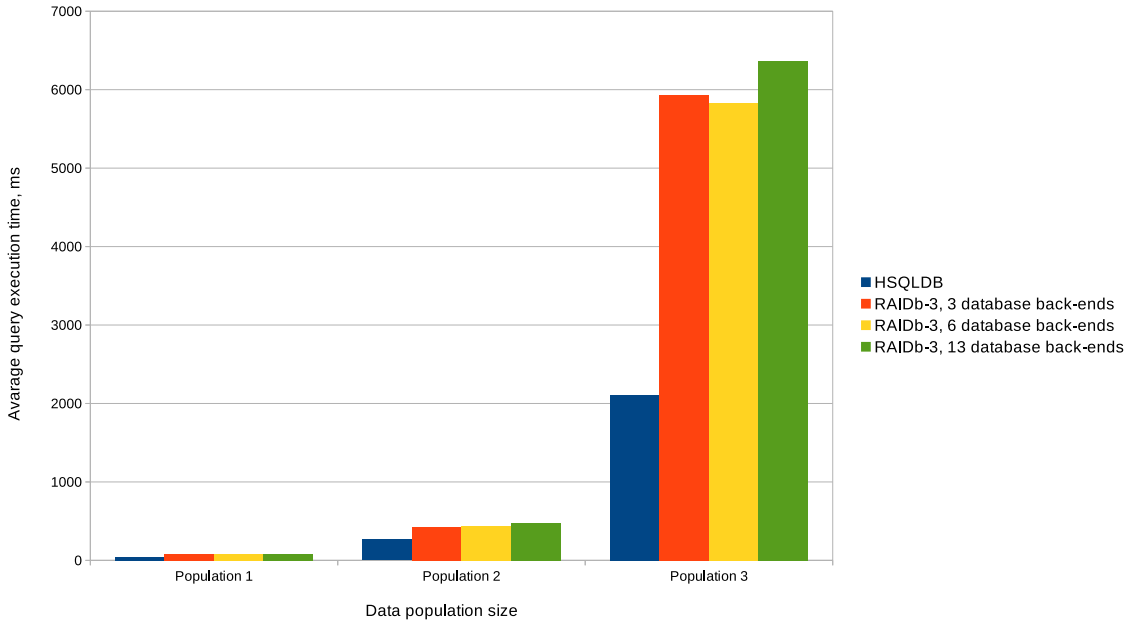

Figure 4.3: Summarized results for non-JOIN Query Z

However, Query Z, which returns a large result set exhibits consistently weaker results than those for the standalone HSQLDB. In order to study this phenomena, we introduced extensive instrumentation to Sequoia's logging subsystem. We extensively analyzed debug logs and, ultimately, we ran several profiler sessions. Our hypothesis on results for non-JOIN queries is the following. The RAIDb-3 controller introduces several overheads when compared to a standalone DBMS, including:

1. Load-balancer overhead associated with the process of selecting the appropriate database back-end.

2. `RaControllerResultSet` overhead due to synchronization and aggregation (including dynamic memory allocation).

3. Additional query parsing and processing.

However, results of our analysis consistently showed that total overhead of the RAIDb-3 implementation does not exceed 20% of total non-JOIN query execution time. Ultimately, we conclude that the most probable source of performance deterioration for queries returning large result sets is Sequoia's proprietary JDBC driver. The driver that is responsible for serialization/de-serialization of the requests, as well as for communicating with other JDBC entities, does not deem to scale effectively, leading to a performance *bottleneck*.

Unfortunately, due to the complexity of the JDBC driver, and the fact that driver design is not in itself a primary focus of this research, we did not investigate the issue of improving scalability of the Sequoia's JDBC driver. However, this might be worth exploring in a future project.

4.3.2 JOIN queries

In Chapter 3.6, we explained that handling JOIN queries is considerably different than handling non-JOIN queries. Therefore, we examine this sort of query in detail in this section.

As we stated earlier, RAIDb-3 currently only supports JOINs between two tables. Furthermore, a JOIN-predicate may only be defined between the primary key of the first table and the foreign key of the second table. Hence, in our evaluation we consider two tables - the `CUSTOMER` table defined in Table 4.2 and the `INVOICE` table shown in Table 4.10.

Table 4.10: Structure of table `INVOICE`

Column name	Column type	Constraint type
InvoiceId	INTEGER	Primary key
InvoiceCustomerId	INTEGER	Foreign key, references column CustomerId from table Customer
InvoiceTotal	DOUBLE	
InvoiceTimeStamp	TIMESTAMP	

The table is created by issuing the SQL statement shown in Listing 4.6.

Listing 4.6: Statement for creating table `INVOICE`

```
CREATE TABLE Invoice(InvoiceId INTEGER GENERATED ALWAYS AS
    IDENTITY PRIMARY KEY, InvoiceCustomerId INTEGER FOREIGN
    KEY REFERENCES Customer(CustomerId), InvoiceTotal DOUBLE,
    InvoiceTimeStamp TIMESTAMP);
```

Both tables are populated with random data from the script generated by SqlGen. An example of the `INSERT` statement for the `CUSTOMER` table is listed in Listing 4.2. An example of the `INSERT` statement for the `INVOICE` table is shown in Listing 4.7.

Listing 4.7: Sample of statement used for populating the `INVOICE` table

```
INSERT INTO Invoice VALUES(DEFAULT, 27, 16.39, '2012-01-23␣
    09:17:38');
```

As with non-join query evaluation, we examine three data population sizes:

1. *Population 1*: 10k records in the table `CUSTOMER` and 100k records in the table `INVOICE`.

2. *Population 2*: 100k records in the table `CUSTOMER` and 1M records in the table `INVOICE`.

3. *Population 3*: 1M records in the table `CUSTOMER` and 10M records in the table `INVOICE`.

Additionally, by varying the ratio parameter in SqlGen, we further split each population into three sub-groups for which the ratio R can assume the values: 1.0, 0.5 and 0.01. The parameter R defines the size of the sub-set which is randomly (as per uniform distribution) filled with primary keys from the `CUSTOMER` table. This sub-set is then used to randomly (as per uniform distribution) pick values for the foreign key in the `INVOICE` table. In essence, ratio allows us to vary the size of the JOIN result based on source data, and not only selection predicates.

Ultimately, we devised three distinct JOIN `SELECT` queries. The queries vary in the degree of data discrimination and, as such, produce results of different sizes for the same input. We refer to these queries as Query L, Query M and Query N. The queries are shown in Listings 4.8, 4.9, and 4.10.

Listing 4.8: Query L

```
SELECT * FROM Invoice, Customer WHERE
    Invoice.InvoiceCustomerId=Customer.CustomerId AND
    Invoice.InvoiceTotal>120 AND
    Invoice.InvoiceTimeStamp>='2016-06-01 00:00:00' AND
    Invoice.InvoiceTimeStamp<='2017-06-30 23:59:59' AND
    Customer.CustomerAge>=30 AND
    Customer.CustomerLocation='Saskatoon' AND
    Customer.CustomerSex='M';
```

Listing 4.9: Query M

```
SELECT * FROM Invoice, Customer WHERE
    Invoice.InvoiceCustomerId=Customer.CustomerId AND
    Invoice.InvoiceTotal>105 AND Customer.CustomerAge>30 AND
    Customer.CustomerAge<=39 AND Customer.CustomerSex='M' AND
    Customer.CustomerProvince='ON';
```

Listing 4.10: Query N

```
SELECT * FROM Invoice, Customer WHERE
    Invoice.InvoiceCustomerId=Customer.CustomerId AND
    Invoice.InvoiceTotal<80 AND
    Customer.CustomerProvince='QC';
```

Similar to the non-JOIN case, we first verify the validity of the results returned by the RAIDb-3 cluster for each query and population. Therefore, we compare results produced by all tested RAIDb-3 configurations (3, 6 and 13 database back-ends) with results returned by a standalone HSQLDB. Generalized validity check results are listed in Table 4.11.

Table 4.11: HSQLDB and Sequoia RAIDb-3 cluster result match

	RAIDb-3, 3 back-ends	RAIDb-3, 6 back-ends	RAIDb-3, 13 back-ends
Match with results from HSQLDB	✓	✓	✓

4.3.2.1 Population 1

Table 4.12 lists result sizes for the tested query types and ratio values. A full set of test results is provided in the Appendix, in Tables A.10 - A.18. Table 4.13 summarizes these results.

4.3.2.2 Population 2

Sizes of the result sets for all respective queries and ratio values are listed in Table 4.14. After performing a sequence of query executions we obtain the quantitative results listed in the appendix, in Tables A.19 - A.27. Table 4.15 summarizes these results.

4.3.2.3 Population 3

Table 4.16 exhibits sizes of the result sets returned for each query type and ratio value. On performing a sequence of query executions we obtain quantitative results listed in the Appendix, in Tables A.28 - A.36. Table 4.17 summarizes these results.

Table 4.12: Size of result sets for test queries

Query type	Result set size		
	R=1.0	R=0.50	R=0.01
L	1	1	23
M	143	133	290
N	4130	4143	2176

Table 4.13: Average JOIN query execution time for tables `CUSTOMER` and `INVOICE` containing Population 1

Database system	Ratio	Query L time, ms	Query M time, ms	Query N time, ms
HSQLDB	R=1.00	33	55	99
	R=0.50	36	53	96
	R=0.01	30	48	73
RAIDb-3, 3 back-ends	R=1.00	684	682	749
	R=0.50	662	651	723
	R=0.01	613	598	661
RAIDb-3, 6 back-ends	R=1.00	811	733	792
	R=0.50	759	715	777
	R=0.01	760	678	742
RAIDb-3, 13 back-ends	R=1.00	1171	1061	1085
	R=0.50	1097	946	983
	R=0.01	1029	860	898

Table 4.14: Size of result sets for test queries

Query type	Result set size		
	R=1.0	R=0.50	R=0.01
L	31	31	35
M	1090	1096	1378
N	41588	41282	44532

Table 4.15: Average JOIN query execution time for tables `CUSTOMER` and `INVOICE` containing Population 2

Database system	Ratio	Query L time, ms	Query M time, ms	Query N time, ms
HSQLDB	R=1.00	246	554	1060
	R=0.50	232	557	1046
	R=0.01	222	448	814
RAIDb-3, 3 back-ends	R=1.00	8990	8143	8888
	R=0.50	8596	7684	8350
	R=0.01	6949	6269	6775
RAIDb-3, 6 back-ends	R=1.00	10633	8781	11450
	R=0.50	9883	9994	11047
	R=0.01	7974	8135	8975
RAIDb-3, 13 back-ends	R=1.00	14554	14785	14736
	R=0.50	14434	14170	14585
	R=0.01	10065	10357	11779

Table 4.16: Size of result sets for test queries

Query type	Result set size		
	R=1.0	R=0.50	R=0.01
L	296	322	323
M	11193	10767	10338
N	414306	412478	402778

Table 4.17: Average JOIN query execution time for tables `CUSTOMER` and `INVOICE` containing Population 3

Database system	Ratio	Query L time, ms	Query M time, ms	Query N time, ms
HSQLDB	R=1.00	2697	8038	14037
	R=0.50	2674	7622	13992
	R=0.01	2547	7274	13252
RAIDb-3, 3 back-ends	R=1.00	128809	102374	108431
	R=0.50	124315	97916	106668
	R=0.01	96925	76915	83890
RAIDb-3, 6 back-ends	R=1.00	167586	135781	139280
	R=0.50	165607	134503	133308
	R=0.01	149590	123409	124248
RAIDb-3, 13 back-ends	R=1.00	204385	181030	177768
	R=0.50	198552	173287	168177
	R=0.01	184024	146489	153044

4.3.2.4 Discussion

Similar to the case of non-JOIN queries, the validity of results is perfectly consistent among all performed tests. Furthermore, we note that this analysis was repeated multiple times throughout the research program.

In the course of quantitative evaluation we discovered that the JOIN query execution performance of the RAIDb-3 controller is inferior to the results demonstrated by standalone HSQLDB. Furthermore, this observation applies to all tested query structures, populations sizes and RAIDb-3 controller configurations. Below, in Figures 4.4 - Figure 4.6, we provide diagrams that summarize our findings. We note that in these diagrams we only depict results obtained for populations with ratio R=0.50; however, the same patterns are observed for all tested ratios, as per Tables 4.13 - Table 4.17.

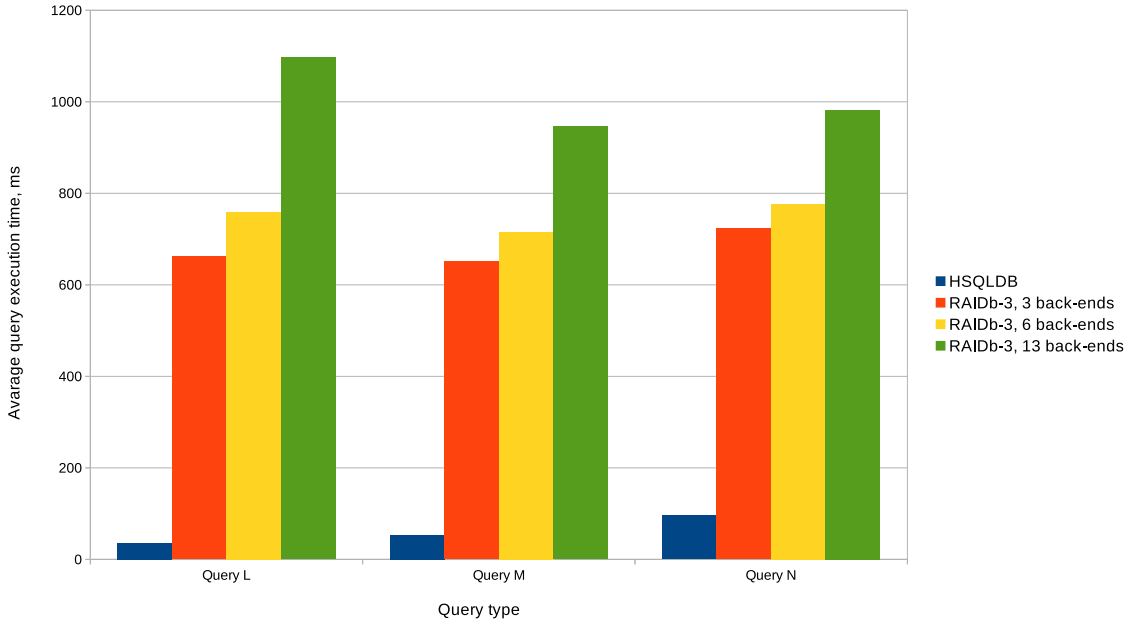

Figure 4.4: Summarized results for JOIN queries for Population 1, R=0.50

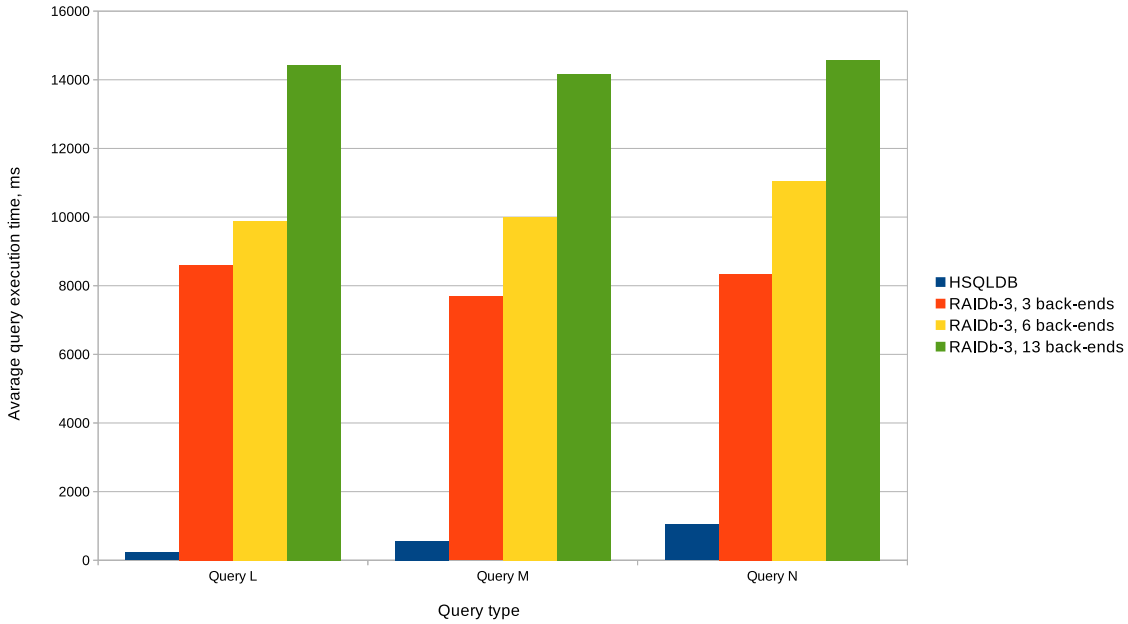

Figure 4.5: Summarized results for JOIN queries for Population 2, R=0.50

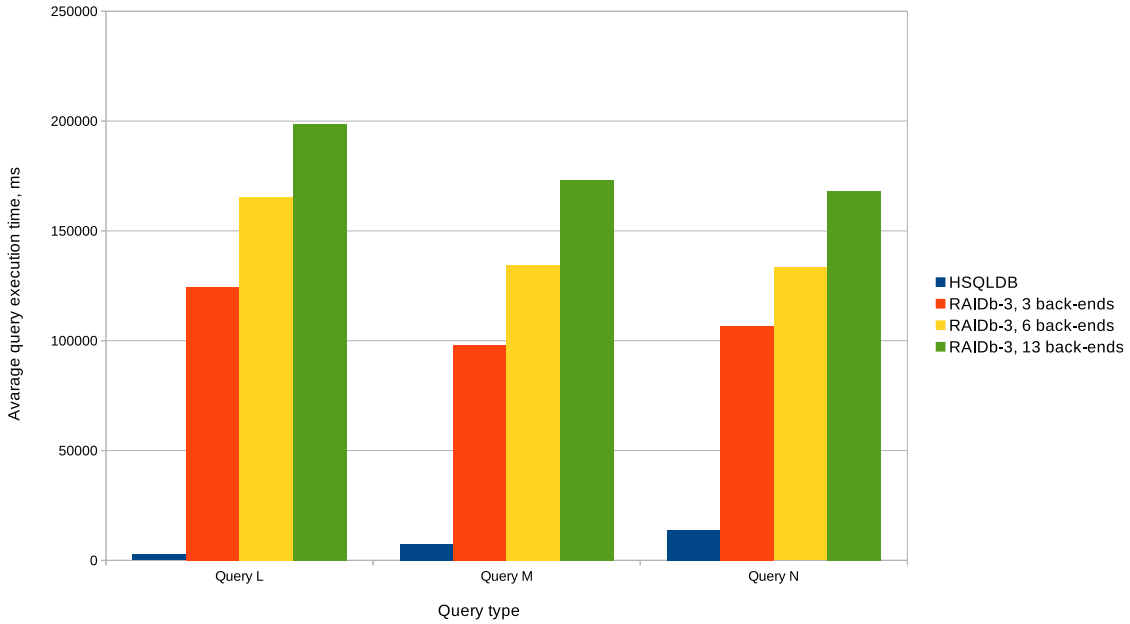

Figure 4.6: Summarized results for JOIN queries for Population 3, R=0.50

Similar to the non-JOIN case, we conducted a full analysis of the results, as well as factors that might affect these results. In order to do so, we again added instrumentation logging to Sequoia and then performed analysis of these logs alongside an examination of profiler sessions. Our findings suggest the following.

1. The Sequoia's JDBC driver. Our findings for the JOIN case are consistent with the conclusions we proposed for non-JOINs in which we determined that Sequoia's JDBC driver has limited scalability in terms of data throughput. However, in the JOIN case, the impact of this issue is aggravated by the fact that the RAIDb-3 model has two stages of data retrieval for JOIN queries. The first stage occurs within the Key Manager and is executed on the Key Tables, while the second stage is performed on the data back-ends. Both stages can involve the transfer of a considerable amount of data and thus performance

deterioration caused by the JDBC driver may have more severe consequences.

2. The `WHERE IN` predicate. As per our findings, the JDBC driver has certain negative impact on overall performance. However, it is not the primary source of performance deterioration in the RAIDb-3 controller. Our analysis clearly indicates that the `WHERE IN` construct utilized in the statement issued by the RAIDb-3 controller to the data back-ends for JOIN queries - such as in Listing 3.35 - is the major source of performance degradation. An argument of the `WHERE IN` clause is an array of integers returned by the Key Manager after executing a JOIN query on the primary and foreign key tables. This array is only limited by the composition of the data population and thus can be rather large (millions of values).

   However, processing of a query with such a large and complex predicate seems to have severe performance implications. To verify this theory, we considered the case of a simple standalone DBMS and prepared a data population and two `SELECT` statements resembling the statement shown in Listing 3.35. The only difference between the two statements is that one does not include a WHERE IN predicate, while the second statement includes 1,000,000 element array as an argument of the WHERE IN. The difference in execution time was by an order of magnitude - 8444 ms for the statement with the WHERE IN, versus 222 ms for the statement without the predicate.

   Indirectly, results of our testing obtained for populations with various ratios support this theory, as data populations with smaller ratios systematically demonstrate better performance, as shown in Figure 4.7. We hypothesize that this is due to the fact that populations with a smaller ratio value produce a smaller

Key Manager JOIN result set. Consequently, the array in the `WHERE IN` predicate is smaller as well and, as such, execution of statements similar to Listing 3.35 consume less time.

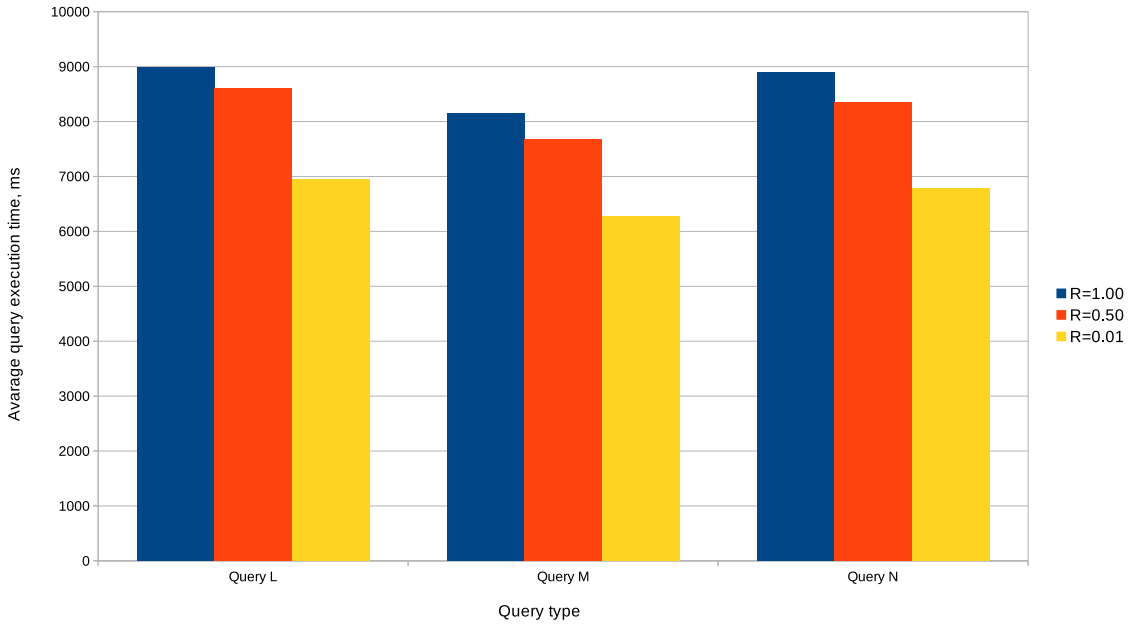

Figure 4.7: Average query execution time for various ratio rates for RAIDb-3 with 3 back-ends and for Population 2

In order to improve overall RAIDb-3 performance for JOIN queries we would suggest the following solutions that might become a foundation for future work.

1. Perform optimization of Sequoia's JDBC driver. This is a rather complex topic and requires extensive research.

2. Develop an approach, possibly with the support of heuristics, that would allow optimal generation of data back-end queries. Here, we suggest the following. For

the cases when a `WHERE IN` predicate argument array is small (it is still necessary to determine the exact size), the RAIDb-3 controller utilizes the existing logic. However, when both the size of the population and the form of the `SELECT` query suggests possible performance loss, the Key Manager should generate data back-end query differently. Specifically, this query should not include `WHERE IN` predicate on the array returned by the Key Manager. Instead a `WHERE IN` logic should be applied to results returned by data back-ends, as a final post-processing stage.

4.4 Conclusion

In this chapter we provided an overview of the existing approaches to the evaluation of DBMSes, as well as, we proposing our methodology to evaluate performance of the RAIDb-3 model. Then, in accordance with the methodology, we prepared a test environment and conducted a series of performance experiments.

Based on the results, we demonstrated that correctness of the results returned by the RAIDb-3 was consistent across all the test cases. Furthermore, we noted that the RAIDb-3 controller exhibits systematically superior performance relative to standalone DBMS for non-JOIN queries, particularly for the case of large populations and a higher degree of distribution (number of data back-ends). However, at the same time, we observed consistently inferior performance demonstrated by the RAIDb-3 controller relative to standard the DBMS for JOIN queries.

Ultimately, we identified a number of factors that might affect performance of the RAIDb-3 controller and suggested possible options for future improvement.

Chapter 5

Conclusions

5.1 Summary

In this work, we presented the design and the implementation of the RAIDb-3 model for full horizontal fragmentation in the Sequoia middle-ware. To that end, we systematized and summarized present advances and developments, as well as new and emerging concepts and solutions in the area of distributed databases. We then applied one specific performance-oriented approach - namely horizontal fragmentation - to the existing Sequoia middle-ware and developed a data distribution model that we called RAIDb-3. Our implementation entirely encapsulates the details of the RAIDb-3 model, thereby allowing transparent integration of horizontal fragmentation into existing data management systems.

This thesis provides an in-depth description of the proposed RAIDb-3 model, as well as outlining limitations of the current prototype. Furthermore, it also demonstrates the viability of the model, using a TPC-H inspired evaluation. Ultimately, we believe that our work may be extended further, particularly in terms of performance optimization, and eventually be adapted for in-production usage.

5.2 Future Work

The research described in this thesis exposes considerable possibilities for future work. As noted earlier, our current implementation is handicapped by certain implementation constraints. Moreover, its performance is not optimal for some workloads and query patterns. Thus, we suggest the following possible directions for future extensions and improvements.

1. Support for queries on more than two tables. In order to achieve this, considerable extension of the logic of the Key Manager will be necessary. To be more specific, for this case the Key Manager should maintain Key Tables for all involved data tables. A more sophisticated query parser will be required as well.

2. Permit JOIN queries to be executed on non primary-foreign key attributes. This functionality, again, will require considerable enhancement of the Key Manager component. Specifically, the Key Manager will need to maintain Key Tables for all columns participating in the JOIN. Here, it is important to note that non primary-foreign key columns are not necessarily of type `INTEGER`; therefore, in order to adhere to the principles of the RAIDb-3 model, hashing of non-integer values will be necessary. Similar to the previous topic, considerable enhancement of the query parser will be required.

3. Optimization of Sequoia's JDBC driver. As we mentioned earlier, Sequoia's JDBC driver does not scale well for large amounts of data. Thus, improving performance and scalability of the JDBC driver might be one of the possible directions for further work.

4. Optimization of JOIN query processing. We provided one of the possible solutions to improve the performance of JOIN queries in Chapter 4.3.2.4.

5. Parallelization of merge and aggregation. Ultimately, to fully exploit distributed computing resources, it is necessary to harness the full power of all nodes throughout the query execution cycle. In particular, we should avoid merging and aggregating partial results on the front-end node, as this represents a potential performance bottleneck. Instead, Sequoia could be extended to allow distributed merging of intermediate results. Such an approach would likely rely on sorting or hashing algorithms.

This is not an exhaustive list; however, we have outlined what we feel to be the most important topics.

5.3 Final Thoughts

This thesis represents considerable research and development effort. We began by reviewing the most significant principles and developments in the area of distributed databases. We then defined the directions and objectives of the work. In the course of completion of the thesis we implemented a robust horizontal partitioning model within Sequoia's RAIDb controller. Given the significance of distributed databases in the modern era, we are convinced that our research represents a meaningful addition to the literature in this area.

Bibliography

[1] U. Röhm A. Dey, A. Fekete. Scalable distributed transactions across heterogeneous stores. *Proceedings of the 2015 ICDE Conference*, pages 125–136, 2015.

[2] E. P. C. Jones Sam Madden A. L. Tatarowicz, C. Curino. Lookup tables: Fine-grained partitioning for distributed databases. *Proceedings of the 2012 ICDE Conference*, pages 102–113, 2012.

[3] S. Weng K. Ren P. Shao D. J. Abadi A. Thomson, T. Diamond. Calvin: Fast distributed transactions for partitioned database systems. *Proceedings of the 2012 ACM SIGMOD Conference*, pages 1–12, 2012.

[4] Amd ryzen. `https://www.amd.com/en/ryzen`.

[5] Y. Zhang S. Madden C. Curino, E. Jones. Schism: a workload-driven approach to database replication and partitioning. *Proceedings of the VLDB Endowment 3 (1-2)*, pages 48–57, 2010.

[6] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 1970.

[7] W. Zwaenepoel E. Cecchet, J. Marguerite. Raidb: Redundant array of inexpensive databases.

[8] S. Madden E. P. Jones, D. J. Abadi. Low overhead concurrency control for partitioned main memory databases. *Proceedings of the 2010 ACM SIGMOD Conference*, pages 603–614, 2010.

[9] A. Salama E. Zamanian, C. Binnig. Locality-aware partitioning in parallel database systems. *Proceedings of the 2015 ACM SIGMOD Conference*, pages 17–30, 2015.

[10] Fedora linux. `https://getfedora.org/`.

[11] S. Babu H. Herodotou, N. Borisov. Query optimization techniques for partitioned tables. *Proceedings of the 2011 ACM SIGMOD Conference*, pages 49–60, 2011.

[12] H.-Y. Hwang and Y.-T. Yu. An analytical method for estimating and interpreting query time. *Proceedings of the 13th VLDB Conference*, 1987.

[13] Hypersql user guide, 2018. `http://hsqldb.org`.

[14] Conflict resolution. `http://www.ittia.com/html/ittia-db-docs/users-guide/replication.html#conflict-resolution`, 2016.

[15] K. Kim D. Kim S. Cha W. Han C. Park H. Na J. Lee, S. Moon. Parallel replication across formats in sap hana for scaling out mixed oltp/olap workloads. *43rd Int'l Conf. on Very Large Data Bases (VLDB)*, pages 1598–1609, 2017.

[16] W. Lin J. Zhou, N. Bruno. Advanced partitioning techniques for massively distributed computation. *Proceedings of the 2012 ACM SIGMOD Conference*, pages 13–24, 2012.

[17] Java. `http://oracle.com/java`.

[18] Java™ platform, standard edition 8 api specification. `https://docs.oracle.com/javase/8/docs/api/`.

[19] Z. Vagena O. Hodson K. Krikellas, S. Elnikety. Strongly consistent replication for a bargain. *Proceedings of the 2010 ICDE Conference*, pages 52–63, 2010.

[20] Netbeans. `http://netbeans.org`.

[21] G. Gibson R. Katz P. Chen, E. Lee and D. Patterson. Raid: High-performance, reliable secondary storage. *ACM Computing Survey*, 1994.

[22] P. J. Pratt and M. Z. Last. *Concepts of Database Management (8 ed.)*. Cengage Learning, 2014.

[23] A. Pavlo M. Stonebraker R. Harding, D. V. Aken. An evaluation of distributed concurrency control. *VLDB 10(05)*, pages 553–564, 2017.

[24] S. Semenova S. Garber R. Marcus, O. Papaemmanouil. Nashdb: An end-to-end economic method for elastic database fragmentation, replication, and provisioning. *Proceedings of the 2018 ICMD Conference*, pages 1253–1267, 2018.

[25] N. Bruno R. Nehme. Automated partitioning design in parallel database systems. *Proceedings of the 2011 ACM SIGMOD Conference*, pages 1137–1148, 2011.

[26] Y.K. Suh R. Zhang S. Currim, R.T. Snodgrass. Dbms metrology: measuring query time. *ACM Transactions on Database Systems*, 2017.

[27] Sequoia user's guide, 2009. `https://sourceforge.net/projects/sequoiadb/`.

[28] P. Subharthi. Database systems performance evaluation techniques.

[29] H.-A. Jacobsen T. Rabl. Query centric partitioning and allocation for partially replicated database systems. *Proceedings of the 2017 ACM SIGMOD Conference*, pages 315–330, 2017.

[30] Tpc benchmark. `http://www.tpc.org`.

[31] Tpc benchmark h. `http://www.tpc.org/tpch`.

[32] A. Pavlo D. Sánchez L. Rudolph X. Yu, Y. Xia and S. Devadas. Sundial: Harmonizing concurrency control and caching in a distributed oltp database management system. *VLDB 11(10)*, pages 1289–1302, 2018.

[33] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems (3rd edition)*. Springer, 2011.

Appendix A

Data tables

Table A.1: Non-JOIN Query X execution times for table `CUSTOMER` with Population 1 (100k records)

Database system	Query execution time, ms									
	1	2	3	4	5	6	7	8	9	10
HSQLDB	61	34	38	34	32	34	28	33	27	30
RAIDb-3, 3 back-ends	101	30	30	28	31	30	31	29	30	28
RAIDb-3, 6 back-ends	89	25	26	27	27	24	25	26	24	26
RAIDb-3, 13 back-ends	77	22	28	23	24	22	20	27	26	22

Table A.2: Non-JOIN Query Y execution times for table `CUSTOMER` with Population 1 (100k records)

Database system	Query execution time, ms									
	1	2	3	4	5	6	7	8	9	10
HSQLDB	26	30	28	28	21	21	25	24	22	25
RAIDb-3, 3 back-ends	26	25	24	24	25	24	24	20	22	25
RAIDb-3, 6 back-ends	20	18	18	17	21	19	22	22	21	19
RAIDb-3, 13 back-ends	18	18	16	19	17	17	19	16	19	19

Table A.3: Non-JOIN Query Z execution times for table `CUSTOMER` with Population 1 (100k records)

Database system	Query execution time, ms									
	1	2	3	4	5	6	7	8	9	10
HSQLDB	52	42	55	51	53	46	45	47	49	38
RAIDb-3, 3 back-ends	117	68	89	80	71	72	70	75	67	74
RAIDb-3, 6 back-ends	116	70	93	90	71	74	70	63	66	65
RAIDb-3, 13 back-ends	107	78	88	101	76	68	67	67	66	69

Table A.4: Non-JOIN Query X execution times for table `CUSTOMER` with Population 2 (1M records)

Database system	Query execution time, ms									
	1	2	3	4	5	6	7	8	9	10
HSQLDB	237	200	211	204	208	207	200	206	202	205
RAIDb-3, 3 back-ends	191	116	119	107	101	105	118	118	103	102
RAIDb-3, 6 back-ends	177	83	84	79	75	85	79	83	82	84
RAIDb-3, 13 back-ends	156	58	66	58	65	62	57	64	59	57

Table A.5: Non-JOIN Query Y execution times for table `CUSTOMER` with Population 2 (1M records)

Database system	Query execution time, ms									
	1	2	3	4	5	6	7	8	9	10
HSQLDB	158	146	153	157	151	158	159	150	145	148
RAIDb-3, 3 back-ends	85	85	82	82	79	96	82	86	81	97
RAIDb-3, 6 back-ends	57	65	65	57	54	61	55	64	67	62
RAIDb-3, 13 back-ends	51	51	44	47	48	43	44	43	45	45

Table A.6: Non-JOIN Query Z execution times for table `CUSTOMER` with Population 2 (1M records)

Database system	Query execution time, ms									
	1	2	3	4	5	6	7	8	9	10
HSQLDB	272	267	305	244	256	254	292	250	259	256
RAIDb-3, 3 back-ends	538	400	444	434	439	404	416	411	408	415
RAIDb-3, 6 back-ends	500	398	435	415	449	434	447	422	398	455
RAIDb-3, 13 back-ends	495	452	453	494	486	509	461	485	464	440

Table A.7: Non-JOIN Query X execution times for table `CUSTOMER` with Population 3 (10M records)

Database system	Query execution time, ms									
	1	2	3	4	5	6	7	8	9	10
HSQLDB	1803	1831	1839	1846	1835	1815	1838	1819	1807	1798
RAIDb-3, 3 back-ends	906	847	843	832	845	832	839	850	842	877
RAIDb-3, 6 back-ends	649	497	539	529	542	558	495	499	505	537
RAIDb-3, 13 back-ends	609	360	367	386	363	362	368	365	379	354

Table A.8: Non-JOIN Query Y execution times for table `CUSTOMER` with Population 3 (10M records)

Database system	Query execution time, ms									
	1	2	3	4	5	6	7	8	9	10
HSQLDB	1651	1602	1599	1830	1638	1643	1614	1598	1645	1652
RAIDb-3, 3 back-ends	884	692	635	639	660	698	640	649	702	728
RAIDb-3, 6 back-ends	436	516	416	445	464	416	428	424	423	439
RAIDb-3, 13 back-ends	335	285	336	309	293	319	296	309	302	304

Table A.9: Non-JOIN Query Z execution times for table `CUSTOMER` with Population 3 (10M records)

Database system	Query execution time, ms									
	1	2	3	4	5	6	7	8	9	10
HSQLDB	2214	1975	2219	2149	2150	2189	2049	1959	2135	2036
RAIDb-3, 3 back-ends	6050	5456	5990	6059	6395	6005	5807	6035	5657	5837
RAIDb-3, 6 back-ends	6006	5295	6464	6068	5865	6008	5576	5639	5963	5395
RAIDb-3, 13 back-ends	6783	6308	6525	6220	6402	6472	6208	6335	6368	6040

Table A.10: JOIN Query L execution times for table `CUSTOMER` and table `INVOICE` with Population 1 and R=1.00

Database system	Query execution time, ms									
	1	2	3	4	5	6	7	8	9	10
HSQLDB	51	31	29	34	27	37	28	33	29	34
RAIDb-3, 3 back-ends	996	678	676	640	686	632	632	649	621	633
RAIDb-3, 6 back-ends	1227	845	821	853	790	789	709	684	678	716
RAIDb-3, 13 back-ends	2156	1190	1109	1055	1042	1068	1020	1024	1008	1039

Table A.11: JOIN Query M execution times for table `CUSTOMER` and table `INVOICE` with Population 1 and R=1.00

Database system	Query execution time, ms									
	1	2	3	4	5	6	7	8	9	10
HSQLDB	63	58	58	51	53	63	56	46	56	47
RAIDb-3, 3 back-ends	686	678	671	687	663	678	674	702	696	687
RAIDb-3, 6 back-ends	802	700	723	808	720	765	705	688	717	705
RAIDb-3, 13 back-ends	1212	1040	1024	1014	1017	1013	1063	1126	1081	1022

Table A.12: JOIN Query N execution times for table `CUSTOMER` and table `INVOICE` with Population 1 and R=1.00

Database system	Query execution time, ms									
	1	2	3	4	5	6	7	8	9	10
HSQLDB	100	98	113	100	100	94	97	106	87	101
RAIDb-3, 3 back-ends	763	761	753	757	757	752	732	757	728	732
RAIDb-3, 6 back-ends	787	801	802	783	761	825	809	793	777	789
RAIDb-3, 13 back-ends	1064	1093	1063	1105	1055	1110	1075	1096	1081	1109

Table A.13: JOIN Query L execution times for table `CUSTOMER` and table `INVOICE` with Population 1 and R=0.50

Database system	Query execution time, ms									
	1	2	3	4	5	6	7	8	9	10
HSQLDB	57	41	37	41	34	29	37	32	30	30
RAIDb-3, 3 back-ends	971	664	650	639	663	609	615	612	606	597
RAIDb-3, 6 back-ends	1286	830	735	691	701	722	699	660	641	634
RAIDb-3, 13 back-ends	2057	1130	1032	1001	949	1033	922	910	924	1015

Table A.14: JOIN Query M execution times for table `CUSTOMER` and table `INVOICE` with Population 1 and R=0.50

Database system	Query execution time, ms									
	1	2	3	4	5	6	7	8	9	10
HSQLDB	62	60	51	47	59	54	50	50	49	52
RAIDb-3, 3 back-ends	665	646	652	636	640	653	656	667	655	647
RAIDb-3, 6 back-ends	771	684	720	704	696	708	761	685	700	723
RAIDb-3, 13 back-ends	1016	957	931	960	923	936	934	951	927	933

Table A.15: JOIN Query N execution times for table `CUSTOMER` and table `INVOICE` with Population 1 and R=0.50

Database system	Query execution time, ms									
	1	2	3	4	5	6	7	8	9	10
HSQLDB	100	95	88	101	94	90	95	102	102	100
RAIDb-3, 3 back-ends	768	742	729	710	708	720	725	711	734	690
RAIDb-3, 6 back-ends	851	758	760	779	791	745	801	764	743	780
RAIDb-3, 13 back-ends	989	1003	982	991	972	971	982	987	981	979

Table A.16: JOIN Query L execution times for table `CUSTOMER` and table `INVOICE` with Population 1 and R=0.01

Database system	Query execution time, ms									
	1	2	3	4	5	6	7	8	9	10
HSQLDB	38	31	33	25	27	31	37	23	30	31
RAIDb-3, 3 back-ends	922	620	627	597	605	552	550	557	551	550
RAIDb-3, 6 back-ends	1255	750	695	773	676	729	643	673	710	696
RAIDb-3, 13 back-ends	2140	1027	951	879	889	928	843	859	849	929

Table A.17: JOIN Query M execution times for table `CUSTOMER` and table `INVOICE` with Population 1 and R=0.01

Database system	Query execution time, ms									
	1	2	3	4	5	6	7	8	9	10
HSQLDB	53	52	55	45	50	40	52	47	46	46
RAIDb-3, 3 back-ends	626	586	590	600	594	593	593	610	598	599
RAIDb-3, 6 back-ends	749	668	667	645	766	650	696	655	663	626
RAIDb-3, 13 back-ends	880	862	859	857	868	854	869	845	853	860

Table A.18: JOIN Query N execution times for table `CUSTOMER` and table `INVOICE` with Population 1 and R=0.01

Database system	Query execution time, ms									
	1	2	3	4	5	6	7	8	9	10
HSQLDB	80	72	71	80	69	75	72	69	70	72
RAIDb-3, 3 back-ends	696	662	657	666	650	652	661	653	662	653
RAIDb-3, 6 back-ends	736	783	703	818	727	761	720	721	700	756
RAIDb-3, 13 back-ends	875	899	902	898	869	893	929	902	913	900

Table A.19: JOIN Query L execution times for table `CUSTOMER` and table `INVOICE` with Population 2 and R=1.00

Database system	Query execution time, ms									
	1	2	3	4	5	6	7	8	9	10
HSQLDB	255	255	240	255	242	241	251	233	242	247
RAIDb-3, 3 back-ends	11408	11545	8945	9610	8595	7813	9132	7608	7518	7730
RAIDb-3, 6 back-ends	14022	14194	11574	10225	10883	10666	8786	8600	8569	8817
RAIDb-3, 13 back-ends	19876	16611	15467	14952	14704	12634	14218	12260	12539	12282

Table A.20: JOIN Query M execution times for table `CUSTOMER` and table `INVOICE` with Population 2 and R=1.00

Database system	Query execution time, ms									
	1	2	3	4	5	6	7	8	9	10
HSQLDB	544	549	540	547	558	542	578	560	561	566
RAIDb-3, 3 back-ends	8876	7940	8165	8025	7818	7900	8153	8132	8205	8216
RAIDb-3, 6 back-ends	8841	8687	8831	8875	8646	8877	8732	8661	8824	8838
RAIDb-3, 13 back-ends	20246	17003	15575	15856	13401	12784	15051	12702	12279	12959

Table A.21: JOIN Query N execution times for table `CUSTOMER` and table `INVOICE` with Population 2 and R=1.00

Database system	Query execution time, ms									
	1	2	3	4	5	6	7	8	9	10
HSQLDB	1059	1090	1046	1063	1077	1046	1082	1028	1067	1046
RAIDb-3, 3 back-ends	9042	8948	8979	8772	8894	8834	9018	8703	8946	8753
RAIDb-3, 6 back-ends	14604	14621	13356	11919	10418	11138	9363	10297	9444	9344
RAIDb-3, 13 back-ends	20596	17413	16195	15715	12786	13476	12549	12893	12715	13026

Table A.22: JOIN Query L execution times for table `CUSTOMER` and table `INVOICE` with Population 2 and R=0.50

Database system	Query execution time, ms									
	1	2	3	4	5	6	7	8	9	10
HSQLDB	263	242	235	228	226	231	226	220	222	228
RAIDb-3, 3 back-ends	10967	11365	8271	8887	8307	7738	7380	8226	7403	7424
RAIDb-3, 6 back-ends	13233	13597	11292	9346	9523	8499	8689	8356	8119	8182
RAIDb-3, 13 back-ends	19830	16855	15084	14441	14057	13931	11961	11798	14676	11713

Table A.23: JOIN Query M execution times for table `CUSTOMER` and table `INVOICE` with Population 2 and R=0.50

Database system	Query execution time, ms									
	1	2	3	4	5	6	7	8	9	10
HSQLDB	551	565	552	566	548	553	569	554	561	553
RAIDb-3, 3 back-ends	7630	7677	7594	7911	7463	7467	7668	7923	7798	7715
RAIDb-3, 6 back-ends	13129	12599	11412	10815	8655	9087	9177	8180	8558	8336
RAIDb-3, 13 back-ends	19080	16207	15230	15097	14099	13959	12082	12228	11631	12088

Table A.24: JOIN Query N execution times for table `CUSTOMER` and table `INVOICE` with Population 2 and R=0.50

Database system	Query execution time, ms									
	1	2	3	4	5	6	7	8	9	10
HSQLDB	1082	1092	1021	1024	1055	1052	1029	1036	1043	1034
RAIDb-3, 3 back-ends	8451	8616	8193	8386	7981	8405	8302	8478	8308	8387
RAIDb-3, 6 back-ends	14883	14153	12794	10808	10440	10184	9346	10042	8979	8843
RAIDb-3, 13 back-ends	19541	16898	14938	16235	14452	13451	12357	13370	12346	12262

Table A.25: JOIN Query L execution times for table `CUSTOMER` and table `INVOICE` with Population 2 and R=0.01

Database system	Query execution time, ms									
	1	2	3	4	5	6	7	8	9	10
HSQLDB	252	234	220	225	205	221	214	208	225	218
RAIDb-3, 3 back-ends	8784	8297	7266	7374	6047	6843	6787	6017	5969	6109
RAIDb-3, 6 back-ends	10596	9913	8184	7924	7829	8164	6722	7235	6315	6861
RAIDb-3, 13 back-ends	14402	13339	11308	8877	8325	8732	8452	10201	8698	8324

Table A.26: JOIN Query M execution times for table `CUSTOMER` and table `INVOICE` with Population 2 and R=0.01

Database system	Query execution time, ms									
	1	2	3	4	5	6	7	8	9	10
HSQLDB	449	431	461	457	445	464	453	458	439	429
RAIDb-3, 3 back-ends	6508	6252	6340	6391	6105	6273	6149	6249	6296	6129
RAIDb-3, 6 back-ends	11005	9710	7968	7963	8278	7676	6910	7814	7585	6445
RAIDb-3, 13 back-ends	14385	13029	10941	10308	10720	8704	10347	8419	8487	8234

Table A.27: JOIN Query N execution times for table `CUSTOMER` and table `INVOICE` with Population 2 and R=0.01

Database system	Query execution time, ms									
	1	2	3	4	5	6	7	8	9	10
HSQLDB	812	810	805	827	788	796	873	807	822	803
RAIDb-3, 3 back-ends	6772	6799	6834	6762	6730	6846	6735	6778	6738	6757
RAIDb-3, 6 back-ends	11535	11613	9029	8545	8666	8563	7397	8576	7510	8323
RAIDb-3, 13 back-ends	17561	14433	12225	13486	11321	11226	9015	11107	8676	8743

Table A.28: JOIN Query L execution times for table `CUSTOMER` and table `INVOICE` with Population 3 and R=1.00

Database system	Query execution time, ms									
	1	2	3	4	5	6	7	8	9	10
HSQLDB	2762	2673	2685	2662	2703	2727	2691	2660	2687	2721
RAIDb-3, 3 back-ends	202005	155051	152090	130049	115052	115233	110376	98184	105092	104964
RAIDb-3, 6 back-ends	225361	176946	178369	161852	162789	158124	150458	153852	160127	147985
RAIDb-3, 13 back-ends	264852	228046	198524	200451	195658	192698	201200	181685	185369	195368

Table A.29: JOIN Query M execution times for table `CUSTOMER` and table `INVOICE` with Population 3 and R=1.00

Database system	Query execution time, ms									
	1	2	3	4	5	6	7	8	9	10
HSQLDB	8022	7668	7650	7777	7722	7712	7779	7664	10599	7791
RAIDb-3, 3 back-ends	101193	99490	108550	100055	99555	100919	100520	110839	101789	100832
RAIDb-3, 6 back-ends	146662	140533	141989	135875	130874	134984	129785	133256	134856	128997
RAIDb-3, 13 back-ends	198698	194689	190784	178780	175693	177698	185365	169557	171788	167248

Table A.30: JOIN Query N execution times for table `CUSTOMER` and table `INVOICE` with Population 3 and R=1.00

Database system	Query execution time, ms									
	1	2	3	4	5	6	7	8	9	10
HSQLDB	14018	14219	14034	14005	14098	13940	14045	14013	14008	13998
RAIDb-3, 3 back-ends	106028	114089	106107	104539	107028	112924	107349	105855	115293	105105
RAIDb-3, 6 back-ends	157690	140478	144652	132145	144985	139138	135887	130997	134256	132569
RAIDb-3, 13 back-ends	199789	187586	184567	175478	174985	170124	179412	168698	167256	169780

Table A.31: JOIN Query L execution times for table `CUSTOMER` and table `INVOICE` with Population 3 and R=0.50

Database system	Query execution time, ms									
	1	2	3	4	5	6	7	8	9	10
HSQLDB	2842	2657	2655	2641	2658	2784	2630	2636	2618	2625
RAIDb-3, 3 back-ends	192492	153630	136361	131432	124048	105758	102573	100704	93709	102443
RAIDb-3, 6 back-ends	218145	190589	169698	177658	178987	162586	135890	148480	141152	132892
RAIDb-3, 13 back-ends	237698	222489	210278	199587	195085	185468	193781	184697	180987	175457

Table A.32: JOIN Query M execution times for table `CUSTOMER` and table `INVOICE` with Population 3 and R=0.50

Database system	Query execution time, ms									
	1	2	3	4	5	6	7	8	9	10
HSQLDB	8090	7496	7568	7575	7496	7757	7576	7620	7547	7496
RAIDb-3, 3 back-ends	98144	96555	95722	96999	101535	96752	96280	97474	101461	98240
RAIDb-3, 6 back-ends	148745	135732	135138	137892	129478	137698	121585	132745	136098	129987
RAIDb-3, 13 back-ends	195878	192364	184360	169458	174258	176325	161874	159368	155998	162987

Table A.33: JOIN Query N execution times for table `CUSTOMER` and table `INVOICE` with Population 3 and R=0.50

Database system	Query execution time, ms									
	1	2	3	4	5	6	7	8	9	10
HSQLDB	14243	13877	13902	13815	14001	14263	13861	14160	13780	14021
RAIDb-3, 3 back-ends	110163	103550	112870	103639	102206	111973	102956	103416	112629	103282
RAIDb-3, 6 back-ends	157895	142987	140685	133138	135895	111586	138168	121586	136254	114890
RAIDb-3, 13 back-ends	191485	185125	174586	161256	168157	167589	165555	159874	155158	152987

Table A.34: JOIN Query L execution times for table `CUSTOMER` and table `INVOICE` with Population 3 and R=0.01

Database system	Query execution time, ms									
	1	2	3	4	5	6	7	8	9	10
HSQLDB	2712	2527	2498	2569	2559	2512	2496	2493	2539	2565
RAIDb-3, 3 back-ends	141851	117882	106761	98748	93954	83786	83716	86085	81821	74648
RAIDb-3, 6 back-ends	168950	175068	159369	149526	157698	161256	135125	124630	136587	127698
RAIDb-3, 13 back-ends	198256	201125	184368	178258	186789	189785	180214	175451	180001	165998

Table A.35: JOIN Query M execution times for table `CUSTOMER` and table `INVOICE` with Population 3 and R=0.01

Database system	Query execution time, ms									
	1	2	3	4	5	6	7	8	9	10
HSQLDB	7323	7216	7203	7230	7441	7291	7270	7219	7299	7257
RAIDb-3, 3 back-ends	82343	75381	75864	76455	75412	75636	82263	75742	75350	74710
RAIDb-3, 6 back-ends	135856	129878	125698	121854	110699	125864	119789	121698	125974	116785
RAIDb-3, 13 back-ends	185985	175985	161874	157895	161458	155989	152478	153666	14978	144587

Table A.36: JOIN Query N execution times for table `CUSTOMER` and table `INVOICE` with Population 3 and R=0.01

Database system	Query execution time, ms									
	1	2	3	4	5	6	7	8	9	10
HSQLDB	13180	13442	13340	13233	13146	13348	13163	13147	13403	13123
RAIDb-3, 3 back-ends	81804	80623	82254	90352	81229	82082	88984	81727	80502	89351
RAIDb-3, 6 back-ends	138138	125684	121778	129365	119458	128706	121902	117125	120450	119874
RAIDb-3, 13 back-ends	181210	175478	165245	159780	140458	141152	144789	138635	146548	137148