**COMP 418 TME 4**

**Jason Bell 3078931**

**May 1, 2016**

**Part 1**

1. **What are the main components of a database workload description?**

This describes the expected load under which a prospective database will operate. It includes:

The list of all implemented queries (transactions that only read) and their relative frequency. Each query description must list which relations are accessed, what columns they select on, and any joins.

The list of all implemented updates (transactions that write) and their relative frequency, as well as the descriptions provided for queries. Each also lists any selections and joins, whether it is an insert, update, or delete, and which columns of which relations are modified.

For each of these queries and updates, a performance goal. That is, how quickly an operation should be carried out. Perhaps some must be completed as fast as possible, while for others it may be acceptable for there to be a delay.

1. **Discuss the main ideas and merits of the two access control mechanisms: discretionary access control and mandatory access control.**

Discretionary access control provides per-user access rights. The user who creates a database object has all rights on the object (they can read and write). The user can grant specific subsets of those rights to other users. In practice it is the database administrator who has the full rights on the objects, and who decides who receives which rights to certain objects.

Mandatory access control more clearly makes the division between database objects and users. Individual objects a security level or class, and users are assigned clearance to access those security classes.

There are two main differences: flexibility and security. Discretionary access control is more flexible and makes it easier to assign custom rights to a certain user. However, the ability of a user to grant someone access to an object they own is a serious weakness. The more people that can give out rights, the more people there are to trick into giving those rights for malicious reasons (I’d argue that this won’t be an issue as long as the database admin is the only one who can grant said rights). Mandatory access control applies security classes and privileges uniformly and at the decision of the database admin. People can access an object or not at the admin’s discretion, and have no ability to grant rights to other parties.

1. **What are the advantages of a distributed DBMS over a centralized DBMS?**
2. Data can be stored redundantly at multiple locations for greater safety in regards to data loss and so that data is still available if a server goes down.
3. The above also makes data more readily available: there may be a server at a close location that has the needed data, rather than a centralized server that may be on the other side of the planet.
4. The pattern of distribution of data can be tailored to suit demands. For example, perhaps each city in a country has a local database providing quick access to that city’s employees. In the event a city needs to access the data for another city, they can still do so through the other city’s local database.
5. Scalability: as demands increase or new company locations are added, new servers can be added to an established distributed database with relative simplicity: the infrastructure is already established.
6. **Explain fragmentation and replication, and explain their differences in data distribution and updating.**

Fragmentation breaks an object and stores the fragments at different locations to cater to the demands of queries at those locations.

Horizontal fragmenting breaks up the relation by row: two cities may have the same employee table, but in one city the table contains it’s employees, while the table of the other city has the employees that city. Both cities can query and update each other’s employee tables as if it is one table, but have more optimal performance with queries on their local fragment.

Vertical fragmentation breaks up the relation by column, typically by what queries select on at each location. If a location often queries two particular columns, there could be a vertical fragment at that location only containing those two columns. The other columns may be stored elsewhere as needed by the queries at those locations. As before, each sees a single table, but have faster queries and updates when they are performed on their local fragment.

Replication is the creation of copies of a database object at more than one location. This adds the complexity that an update to one replicate must be reflected in the other replicates (whether that’s immediate or “eventually” depends on the needs of the users. “Eventually” may be acceptable if users understand that they may see delays). As described in question 3, this is one of the ways to provide faster access and availability of data.

Replication queries and updates can be synchronized by voting: a minimum number of replicates must be written to in order to ensure that at least one returns the new value on a minimum number of reads (the one with the new value is the one with the newest timestamp). For example, if there are 10 replicates, and 8 must be written to, 3 must be read to ensure that at least one will have the right value.

More simply, one can write to all replicates. This results in slower updates, but faster reads since you only need to read one. Deciding which strategy to use depends on the load: the voting system is more suitable to a system with more writes relative to reads, since there are fewer writes. The latter strategy is better for systems with few writes but many reads.

The previous examples are synchronous methods: the result of a write is seen by all queries with no inconsistencies due old data. In many cases, an asynchronous approach where the system doesn’t worry about consistency may be acceptable. Operations will be faster, since reads don’t wait for a certain number of writes to be done. The downside is that users will see inconsistency until all replicates have been updated. This may be acceptable as long as users expect this behavior.

**Part 2**

1. **You have been hired as database administrator for Athabasca University and the computing services director asked you to tune the following database that is too slow for query processing.**

**The database has two relations:**

**Professor(sin, prof\_name, office\_no, age, gender, specialty, dept\_did)  
Department(did, dept\_name, budget, nbr\_programs, chair\_sin)**

**After examining the application you found that the following queries are the five most common queries in the workload for this university application and that all are roughly equivalent in frequency and importance:**

* **List the names, ages, and offices of professors of a user-specified gender (male or female) who have a user-specified research specialty (e.g., recursive query processing). Assume that the university has a diverse set of faculty members, making it very uncommon for more than a few professors to have the same research specialty.**
* **List all the department information for departments with professors in a user-specified age range.**
* **List the department id, department name, and chairperson name for departments with a user-specified number of majors.**
* **List the lowest budget for a department in the university.**
* **List all the information about professors who are department chairpersons.**

**These queries occur much more frequently than updates, so you should build whatever indexes you need to speed up the queries. However because updates do occur, you should not build any unnecessary indexes that would slow down the updates. Given this information, design a physical schema for the university database that will perform well for the expected workload. In particular, decide which attributes should be indexed and whether each index should be clustered or unclustered. Assume that both B+ trees and hashed indexes are supported by the DBMS and that both single- and multiple-attribute index search keys are permitted.**

1. **Specify your physical design by identifying the attributes you recommend indexing on, indicating whether each index should be clustered or unclustered and whether it should be a B+ tree or a hashed index.**

In every answer I opt to not cluster, since I know nothing about what kinds of updates are performed. I will specify where a clustering would be beneficial, assuming I knew that it wouldn’t hinder updates too severely.

For the first query, even though we are selecting on both gender and speciality, it at first seems that a hash index (because it’s faster for equality) on (gender, speciality) of the Professor table may work well. It likely would, but given that we know that on average there will only be a few professors of a given speciality, having gender in the index is essentially redundant. A hash index on speciality will already return a very small number of professors, at which point it is trivial to drop those professors not of the gender being requested. Given that each query will return a very small number of records, there is no benefit to making the index clustered (especially compared to the adverse affect that clustering will have on updates).

For the second query, we are doing a range selection on age of the Professor table, so this will benefit from a B+ index on the (age) column of that table. We also need the professor’s dept\_did, so we may as well expand the index to (age, dept\_did). This index allows us to do an index only scan on the only columns we need from the Professor table, so clustering is unnecessary: we won’t even be accessing the data file for that part of the query. For the second part, we are getting the department information for the dept\_did from the first part: I am assuming that this means that we are retrieving all fields of the Department relation. This means we won’t benefit from an index only scan, nor will clustering provide any benefit: there is unlikely to be a correlation between age and dept\_did that would result in useful clustering. Therefore, I would simply do a hash index on (did) of the Department table since we want to do an equality selection on that field. At this point it is a matter of performing a join on department id of the data returned from the two parts of the query.

For the third query, we could start with a hash index on (nbr\_programs) of the Department table, since we’re doing an equality selection on this field. This will quickly point to those records with the requested number of programs. We could cluster by nbr\_programs, but again, I am loath to cluster any data without knowing anything about the updates. A hash index on (sin) of the Professor table would allow us to perform a join to get those professors that match chair\_sin from the previous result.

The fourth query would be optimized simply by a B+ index on (budget) of the Department table: the “minimum value” is provided by a range selection. This won’t benefit from clustering, since we are only interested in the budget value, which we can obtain by scanning the index.

The fifth query would use the same indexes as the third, except that we will return professors information rather than department information.

1. **Assume that this workload is to be tuned with an automatic index tuning wizard. Outline the main steps in the algorithm and the set of candidate configurations considered.**

Index tuning wizards have two steps: candidate index selection and enumeration.

In candidate selection, we only accept those indexes which index on a selection with the WHERE clause or which is involved in GROUP BY or ORDER BY. All indexes meet these requirements except for the first index for the second query, that indexes on (age, dept\_did). I chose this index to allow an index-only scan, but dept\_did is not involved in WHERE, GROUP BY, or ORDER BY (it seems strange to me that candidate selection doesn’t consider those indexes that would benefit joins…).

In the second step, the wizard enumerates the candidate indexes and uses heuristics to determine their costs, and decide while indexes will most benefit a given query.

1. **You are a DBA for the accounting firm AFS. The firm is a merging of two previous companies so they need to implement a new human resource system. You have created a relation about employees as follows:**

**Employee(ename, dept, salary, address, telephone, job).**

**For authorization reasons, the HR director asked you to create two views for her:**

* **EmployeeNames that displays only the employee name (ename).**
* **DeptInfo that lists the average salary for each department (dept, avgsalary).**

1. **Show the view definition statements for EmployeeNames and DeptInfo.**

CREATE VIEW EmployeeNames (ename)

AS SELECT E.ename

FROM Employee E

CREATE VIEW DeptInfo(avgsalary)

AS SELECT dept, AVG(salary)

FROM Employee

GROUP BY dept

1. **What privileges should be granted to a user who needs to know only average department salaries for the HR and CS departments?**

The user should only have a privilege on the DeptInfo view. The user does not need a privilege on the EmployeeNames view. This will allow the user to get the average, but not determine the confidential salaries of individual employees (unless there’s a single employee of course).

However, the view as described shows the average salaries for all departments, and so will allow the user to see departments other than HR or CS. An additional view would be needed that has a WHERE clause that selects on dept:

CREATE VIEW DeptInfoHRCS(avgsalary)

AS SELECT dept, AVG(salary)

FROM Employee

WHERE dept == ‘HR’ OR dept == ‘CS’

GROUP BY dept

1. **The HR director is taking an extended vacation, and to make sure that emergencies can be handled, she want to authorize her boss Joe to read and modify the Employees relation and the EmployeeNames relation (and Joe must be able to delegate authority, because he is too far up the management hierarchy to do any actual work). Show the appropriate SQL statements. Can Joe read the DeptInfo view?**

GRANT SELECT, UPDATE, INSERT, DELETE on Employees, EmployeeNames TO Joe WITH GRANT OPTION

Joe can read the DeptInfo because he has full privileges on the underlying relation: Employees.

1. After returning from her (wonderful) vacation, she found a note from Joe, indicating that he authorized his secretary Mike to read the Employees relation. She asked you to revoke Mike’s SELECT privilege on Employees, but she does not want to revoke the rights she gave to Joe, even temporarily. Can you do this in SQL?

Assuming that Joe did not grant SELECT to Mike with GRANT OPTION, he can simply do the following since he is the ultimate owner of the relations:

REVOKE SELECT ON Employees FROM Mike RESTRICT

If Joe used GRANT OPTION, Mike may have granted others the privilege, so dependent privileges must be deleted with cascade:

REVOKE SELECT ON Employees FROM Mike CASCADE

**Part 3**

I have been a research assistant for Dr. Kumar for the last two years, and this very week I am starting work on two different projects that use Apache Cassandra as the data store. I have used MySQL extensively, but have yet to use Cassandra, so will use this report as an opportunity to learn about that DBMS. Therefore, consider this report to be a review of the strengths and applications of Cassandra, with the target audience being inexperienced students such as myself.

**Abstract**

Apache Cassandra was first developed by Facebook, and since being made open source in 2008 has become the de facto standard for distributed databases. Apache Cassandra offers clustered scaling, reliability, configurable consistency, fault tolerance, and a flexible data model.

This paper reviews and discusses these features of Apache Cassandra and the advantages that they provide to the world of Big Data.

**Introduction**

Apache Cassandra is an asynchronous NoSQL distributed database that was originally developed by Facebook and has received extensive adoption [1]. NoSQL databases are designed so that data is easily distributed across nodes in the cluster, and new nodes can be added easily. NoSQL databases are also asynchronous: updates propagate “eventually”, rather than immediately.

Apache Cassandra has achieved wide adoption through it’s robust support for scaling, reliability, configurable consistency, fault tolerance, and flexible data model. These traits make Apache Cassandra ideally suited as an ingestion service data store. Data throughput increases linearly as new nodes are added to the cluster [2]. It can handle a large throughput of incoming data and an ingestion service does not need to have writes propagate immediately: in a “big data” context, it only matter that the nodes update eventually. Apache Cassandra has particularly been adopted by the academic community as it is free and open source, and because it is implemented in and strongly supports Java: a highly ubiquitous language in academia.

**Replication and Consistency**

Data is distributed across nodes through replication, allowing for operation to continue if a node fails [3]. The manner of data distribution via replication is highly customizable by node and by data center. For a single node or data center, a SimpleStrategy configuration is provided that replicates data in round-robin fashion without considering physical locations, if multiple nodes exist at that data center. The more advanced NetworkTopologyStrategy considers node locations across multiple data centers and allows the user to configure how many replicas to store at each data center [4].

Apache Cassandra offers eventual consistency, which would make it inappropriate where updates must be atomic, or atomic updates can be provided, depending on configuration. Write consistency is highly configurable with the following levels, where a “quorum” is a number of replicas determined by the replication factors of the data centers [5]:

Quorum = (sum of replication factors across datacenters / 2) + 1 rounded down [5]

For example, suppose that a single data center has a replication factor of 4. This will result in a quorum of 3. If there are two data centers with a replication factor of 3, a quorum will be 4. A quorum determines both the behavior of reads and writes and the cluster’s tolerance for failed nodes. In the case of the two data centers with a replication factor of 3, there are 6 replicas and the system can operate as long as no more than 2 fail, assuming that a quorum configuration is used [5].

|  |  |
| --- | --- |
| **Level** | **Description** |
| ALL | A write must update all replicas |
| EACH\_QUORUM | A write must update a quorum of replicas at all data centers |
| QUORUM | A write must update a quorum of replicas at any data center |
| LOCAL\_QUORUM | A write must update a quorum of replicas at the local data center. |
| ONE | A write must update one or more replica |
| TWO | A write must update two or more replicas |
| THREE | A write must update three or more replicas |
| LOCAL\_ONE | A write must update one or more replica at the local data center |
| ANY | A write must update at least one replica |

Table 1. Write consistency levels [5].

These write consistency levels are paired with similar read consistency levels:

|  |  |
| --- | --- |
| **Level** | **Description** |
| ALL | A read will read all replicas and return the most recent |
| EACH\_QUORUM | A read will read a quorum of replicas in all data centers and return the most recent |
| QUORUM | A read will read a quorum of replicas from any data center |
| LOCAL\_QUORUM | A read will read a quorum of replicas from the local data center |
| ONE | A read will read the closest single replica |
| TWO | A read will read the closest two replicas and return the most recent |
| THREE | A read will read the closest three replicas and return the most recent |
| LOCAL\_ONE | A read will return the closet replica to the local data center |

Table 2. Read consistency levels [5]

Combining these read and write levels results in highly configurable consistency levels. A write consistency of ALL combined with a read consistency of ONE results in complete consistency and the appearance of atomic transactions to the user, at the cost of slower writes as the size of the cluster increases. At the opposite extreme, a write consistency of ANY combined with a read consistency of ALL will result in a cluster that has very rapid writes, and complete consistency, at the cost of slow reads: a read must check all replicas to find the most recent. A write consistency of ANY combined with a read consistency of ONE will be the fastest configuration, but also the least consistent: the likelihood of a read returning the most recent value would be a function of the size of the cluster. This illustrates how highly configurable Apache Cassandra’s consistency is: choose the combination of consistency levels that is appropriate for the use case.

In cases where the ALL write configuration is not used, Apache Cassandra makes use of reads to update those replicas that are not up to date and ensure eventual consistency. When a read completes, the system uses the returned value to update all replicas to that value. Since this happens after the read is complete, it won’t impact the performance of that read. The result is that replicas are not updated continuously: they are either one of the replicas that are updated in the most recent write, or get updated to the most recent value after a read occurs [6].

**Data Format**

As a NoSQL database, Apache Cassandra does not store data as traditional relational tables. Apache Cassandra treats tables as “keyspaces”: the outermost indexed data container. A keyspace can contain any number of columns, or columns grouped into “column families” on a column key. These columns can be combined in any combination to produce a row, which is indexed by a row key. In this manner, an Apache Cassandra “table” is a map where rows and columns can exist in any combination [7].

The flexibility provided by this row-column mapping is that Apache Cassandra does not rely upon a pre-defined schema. Any sort of data can be added to the database and the columns and rows will adapt to suit it. This makes Apache Cassandra highly suitable for ingestion and storage of unstructured big data.

**Scalability and Fault Tolerance**

Because there is no master server, and all nodes share responsibility evenly, Apache Cassandra scales linearly and without interruption. When nodes are added or removed, responsibility for data is evenly redistributed. Each node has the same replicas of data, and is responsible for serving a portion of it. If a node is removed or fails, the remaining nodes will have that node’s responsibilities evenly distributed amongst them, preventing total failure of the system. If a node is added, responsibilities are transferred to it [8]. The result is a system that is simple to expand to increase demand or respond to failure.

However, as the system scales it is important to consider the current read and write levels: a larger system will have a larger quorum, and this may impact performance. Therefore, adjusting these levels may be necessary.

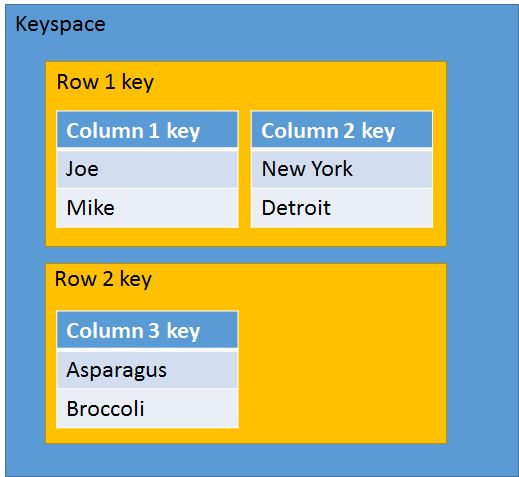


Figure 1. Hierarchy of key index mapping in Apache Cassandra.

**Hadoop Integration**

Apache Hadoop is a suite of tools that provide distributed, scalable, and fault resistant processing of large data sets [9]. Therefore, the distributed, scalable, and fault resistant data storage provided by Apache Cassandra pairs well Apache Hadoop. At the time of writing, Apache Cassandra supports retrieval of data by Apache MapReduce, Pig, and Hive [10].

Apache MapReduce is Hadoop’s MapReduce tool for processing often multi-terabyte data sets. MapReduce tools provide a map function that splits up or classifies the data, and passes these subsets of data to a reduce algorithm that performs inferences. In this manner, structure is mapped onto unstructured data to ease analysis [11]. For example, for census data, the map function may split up the data by demographic, then for each demographic the reduce function may perform statistical analysis.

Like Apache Cassandra, MapReduce is designed to scale without interruption and to be fault tolerant [11]. A large Apache Cassandra cluster is highly appropriate as a scalable and fault tolerant data store for MapReduce.

Apache Pig and Hive behave similarly and in tandem to Apache MapReduce. Apache Pig provides a high level language with which to implement data processing algorithms, that Pig then deploys as MapReduce tasks [12]. Similarly, Hive provides a SQL-like query language for use by MapReduce [13].

**Applications**

While many companies use Apache Cassandra, for a variety of purposes, the most common theme appears to be bulk ingestion of unstructured data. In other cases, where data is structured Apache Cassandra offers fewer advantages, but still offers scalability and reliability [14].

The best example that fits the very definition of “big data” is that Apache Cassandra is used to store data from the CERN Large Hadron Collider: both the physics detected and tracking of the behavior of the devices themselves. CERN describes a number of advantages that Apache Cassandra offers them:

* 1. Large Hadron Collider data is generated in massive but irregular bursts. Apache Cassandra is able to scale to accommodate these bursts.
  2. Large Hadron Collider data is continually changing and evolving, which makes storing them in a schema-based relational database challenging.
  3. Large Hadron Collider is collected over time and is timestamps. Apache Cassandra eases storage and retrieval of these time ordered data by allowing them to use timestamps as row keys [15].

As mentioned to the marker before the start of this report, the author is a research assistant with Athabasca University who has just joined two projects that make use of Apache Cassandra. Unfortunately, one is covered by a non-disclosure agreement and can not be discussed.

The other project is entitled “MORPH” (full name to be determined) under the management of Dr. Sabine Graf and will be a system that ingests interaction data from Athabasca University’s Moodle installation. This data is expected to be large in volume and highly unstructured: for example, series of timestamped forum posts and snapshots of programming code. Apache Cassandra is highly suited to this sort of bulk ingestion of data for later inspection.

Once stored, the data will be input into a machine learning system such as a neural network in order to make inferences about student interactions: do students with a higher degree of interaction perform better? Did students who interacted more with a particular learning object do better? Did they do worse? Neural networks typically have predefined input nodes that accept specific inputs: Apache Cassandra’s support for MapReduce systems will greatly facilitate this process by structuring the data in a way that the neural network understands. Put simply, the data flow will flow as:

1. raw unstructured data from Moodle sensors
2. storage in an Apache Cassandra database
3. processed a more structured form in the map stage of MapReduce
4. processed in the reduce stage to a form that’s ready to enter the neural network
5. processed by the neural network to create inferences about student interaction
6. presented back to students and university officials as a student interaction report.

**Conclusion**

Apache Cassandra is a database system that has received wide adoption by virtue of it’s scalability, reliability, and flexibility. Scalability is provided by it’s ability to have new nodes added to the cluster easily and without interruption. Reliability is provided through it’s replication of data and absence of a master server: there is no single point of failure. Flexibility is provided through it’s schema-less storage of unstructured data in flexible maps of indexed rows and columns. Further flexibility is provided through it’s many configuration options: consistency of data can be adjusted to whatever the requirements are: from fully consistent to extremely inconsistent data that eventually becomes consistent. Finally, support for Apache Hadoop systems makes Apache Cassandra an excellent data store for distributed and reliable data processing.

It is for these reasons that Apache Cassandra is widely used for reliable and scalable storage of data, whether that data is structured or unstructured. Apache Cassandra is particularly suited for Big Data work, where large batches of unstructured data must be stored and then passed to a MapReduce system.

**Citations**

[1] Apache Cassandra. *Cassandra*. Available online: <http://cassandra.apache.org/>. 2015.

[2] Rabi T, Sadoghi M, and Jacobson, H. *Solving Big Data Challenges for Enterprise Application Performance Management*. Proceedings of the VLDB Endowment vol. 5 Issue 12, Aug 2012.

[3] Apache Cassandra. Available online: <http://immagic.com/eLibrary/ARCHIVES/GENERAL/WIKIPEDI/W120911A.pdf>. 2012.

[4] Datastax. *Apache Cassandra 2.0 - Data Replication*. Available online: <https://docs.datastax.com/en/cassandra/2.0/cassandra/architecture/architectureDataDistributeReplication_c.html>. 2016.

[5] Datastax. *Apache Cassandra 2.0 – Configuring data consistency*. Available online: <http://docs.datastax.com/en/cassandra/2.0/cassandra/dml/dml_config_consistency_c.html>. 2016.

[6] Cassandra Wiki. *Read Repair*. Available online: <https://wiki.apache.org/cassandra/ReadRepair>. 2013.

[8] Techblogs. *Cassandra: the right data store for scalability, performance, availability, and maintainability*. Available online: <http://www.rapidvaluesolutions.com/tech_blog/cassandra-the-right-data-store-for-scalability-performance-availability-and-maintainability/>. 2015.

[9] Apache Hadoop. *Hadoop*. Available online: <http://hadoop.apache.org/>. 2014.

[10] Cassandra Wiki. *Hadoop support*. Available online: <https://wiki.apache.org/cassandra/HadoopSupport>. 2014.

[11] Apache Hadoop. *MapReduce tutorial*. Available online: <https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html#Overview>. 2008.

[12] Apache Hadoop. *Welcome to Apache Pig!* Available online: <http://pig.apache.org/>. 2012.

[13] Apache Hadoop. Apache Hive. Available online: <http://hive.apache.org/>. 2014.

[14] Planet Cassandra. Apache Cassandra use cases. Available online: <http://www.planetcassandra.org/apache-cassandra-use-cases/>. 2015.

[15] Planet Cassandra. Cassandra at CERN. Available online: <http://www.planetcassandra.org/blog/cassandra-at-cern-large-hadron-collider/>. 2013.