Topic 5.4: NoSQL Database Implementations

In this topic, we explore database implementation using NoSQL databases. As mentioned in Topic 1.4, NoSQL is a broad and diverse collection of technologies. It is not possible to provide in-depth coverage of the entire spectrum of NoSQL technology implementations in an introductory course such as this one. Therefore, this topic provides a much higher level overview of the implementation considerations for representative NoSQL technologies.

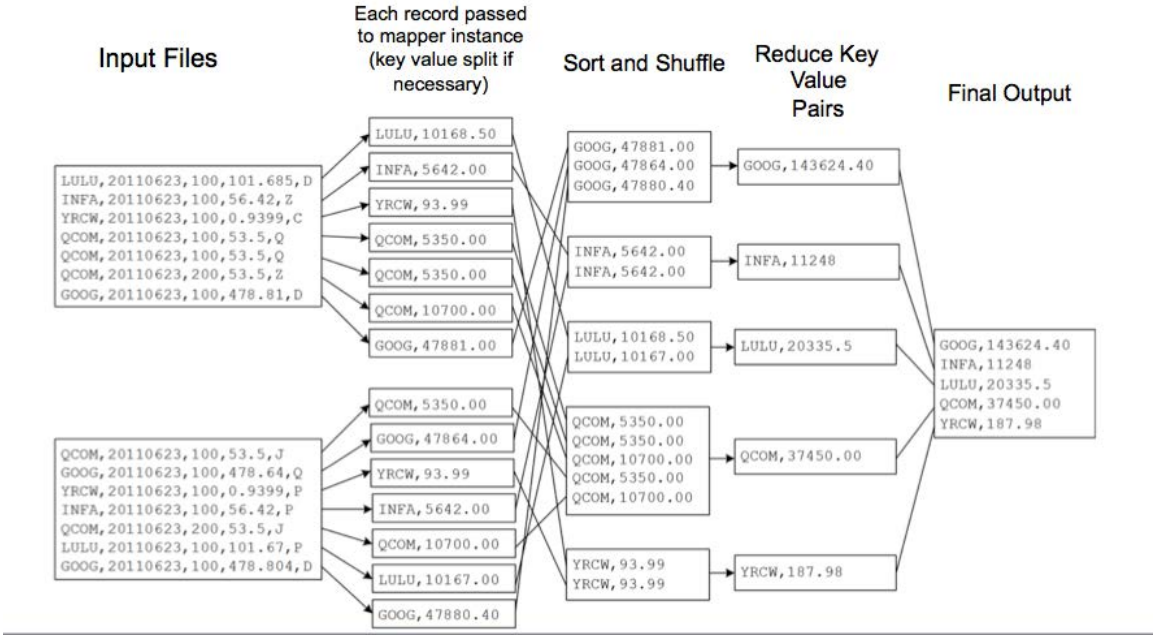
Hadoop Implementation

Apache Hadoop is an open-source framework for distributed storage and processing of unstructured data on computer clusters constructed using commodity hardware. The core components of Hadoop are the Hadoop Distributed File System (HDFS) and MapReduce. The HDFS manages the distribution and storage of data files on the cluster server in a fault-tolerant manner such that no data is lost when a node (computer) in the cluster fails. Other file systems such as Amazon S3 are also supported. MapReduce takes a bit more explanation.

MapReduce

MapReduce is a framework for breaking files into smaller blocks and distributing the blocks across the cluster. This enables parallel processing, which is essential for efficiently handling large volumes of data (so-called "big data"). The mapping processes separate data file records into key-value pairs, and prepares the data for the reduce process. The reduce process combines key-value pairs in order to derive one record (block) per key, and the results are stored as a file that is prepared for analysis.

Here is a diagram of a representative MapReduce process.



In this example, the input files contain historical stock trade transactions. The input files are comma-separated records containing the following fields: Stock Symbol, Trade Date (YYYYMMDD format), the Number of Shares traded, Price Per Share, and Stock Class. In this example, we want to find the total value of the shares traded (calculated by multiplying the number of shares by the price per share), for each stock symbol, regardless of the date or stock class.

The first mapping step extracts the Stock Symbol field and calculates the Total Value field. Each record from the source files is handled by an independent mapper instance. In cases where input file records contain multiple occurrences of the desired fields, a key value split mapper task would also be required to produce multiple key-value pairs from each input record. (In this case, this is unnecessary because each input record contains only one stock trade transaction, and thus only one combination of Stock Symbol, Number of Shares, and Price Per Share.

The next step sorts the key-value pairs by the keys, and "shuffles" keys to individual nodes (computers) so that each node contains the key-value blocks for just one key value.

The Reducer process then performs some form of aggregation of the blocks for each key value, producing a single output record for each key. In this example, we are summing the values for each key, but in other cases, we might want to find the minimum, maximum, or average values, or we might simply want to count how many records were found for each key.

The Hadoop Ecosystem

The Hadoop Ecosystem is a collection of additional software packages that can be installed on top of or alongside Hadoop. Configuration of a Hadoop installation largely depends on the packages selected for it. Here are some of the more popular packages:

- **Hive** is a data warehouse infrastructure component that supports analysis of large datasets stored in Hadoop's HDFS, or compatible file systems such as Amazon S3. Hive Hadoop was founded by Jeff Hammerbacher while he was working at Facebook. Hive incudes an SQL-like query language (HiveQL) that can be used to mine large amounts of data, and transform queries into MapReduce, Apache Tez, and Spark jobs. Other features of Hive include:
    - Indexing, primarily using bitmap indexes, although others are planned.
    - Support for file types beyond the text files supported by HDFS, including HBase, RCFile, ORC, and others.
    - Metadata storage in an RDBMS, such as MySQL or Apache Derby, to speed semantic checking of data.
    - Direct support for compressed data stored in HDFS using various algorithms.
    - Built-in user defined functions (UDFs) for supporting dates, strings, and additional data mining tools.
- **Pig** is a high-level tool for creating MapReduce programs. Pig was originally developed at Yahoo Research around 2006, and moved into the Apache Software Foundation in 2007. *Pig Latin* is the language used in Pig. Pig Latin is a pipeline language, similar to writing extract, transform, and load steps in ETL. In contrast, SQL is considered a declarative language.

Other features of Pig include:

- Extensions to Pig Latin with user defined functions (UDFs) written in Java, Python, JavaScript, Ruby, or Groovy.
- The ability to store data at any point in the process.
- Support for major data operations such as ordering, filtering and joins.
- A declared execution plan. In SQL, the DBMS query optimizer evaluates each query and chooses and execution plan, so the SQL programmer has less control over operations such as joins. In Pig Latin, the programmer declares how the joins and other operations will be executed.
- The ability to split data pipelines so that data need not be processed in a strictly sequential manner.
- Support for a wide range of data types that are not present in MapReduce.

- **Apache YARN** (Yet Another Resource Negotiator) is a resource-management platform used for managing computing resources across cluster servers, and for scheduling applications.
- **Apache Spark** is a data analytics cluster computing framework originally developed in the AMPLab at UC Berkeley. Spark builds on top of Hadoop HDFS, providing an alternative to

Hadoop MapReduce that is easier to use and offers performance that is 10 times faster for certain applications.

- **JAQL** is a declarative programming language designed for processing large volumes of structured, semi-structured, and unstructured data. As the name implies, JAQL is primarily used for handling data stored in JSON documents, but it is capable of handling other types of data such as XML, comma-separated value (CSV) data, and flat files.

Columnar Databases

This section discusses some of the physical design considerations for columnar databases. There is a wide variety of columnar technology on the market, each product having a unique set of physical characteristics. Some of the most popular products in this category are Apache HBase, Apache Cassandra, Hypertable, and HP Vertica. (Some would argue that HP Vertica does not qualify as a NoSQL database because it uses standard SQL with language extensions, but it is clearly a columnar database, so it fits better here than in the relational category.)

As noted in Topic 1.4, columnar databases organize and process data by columns, as opposed to relational databases, which organize and store data by row. For example, consider the Employee table as it would be stored in a relational DBMS:

Employee Table: Relational DBMS

ROWID	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	SALARY	COMMISSION_PCT
1	173	Sundita	Kumar	6100	0.1
2	172	Elizabeth	Bates	7300	0.15
3	171	William	Smith	7400	0.15
4	170	Tayler	Fox	9600	0.2
5	169	Harrison	Bloom	10000	0.2
6	168	Lisa	Ozer	11500	0.25
7	150	Peter	Tucker	10000	0.3
8	151	David	Bernstein	9500	0.25
9	149	Eleni	Zlotkey	10500	0.2

10	145	John	Russell	14000	0.4
11	147	Alberto	Errazuriz	12000	0.3
12	146	Karen	Partners	13500	0.3
13	152	Peter	Hall	9000	0.25
14	155	Oliver	Tuvault	7000	0.15
15	153	Christopher	Olsen	8000	0.2
16	148	Gerald	Cambrault	11000	0.3
17	154	Nanette	Cambrault	7500	0.2
18	156	Janette	King	10000	0.35
19	161	Sarath	Sewall	7000	0.25
20	160	Louise	Doran	7500	0.3
21	159	Lindsey	Smith	8000	0.3
22	158	Allan	McEwen	9000	0.35
23	157	Patrick	Sully	9500	0.35
24	162	Clara	Vishney	10500	0.25
25	163	Danielle	Greene	9500	0.15
26	164	Mattea	Marvins	7200	0.1
27	165	David	Lee	6800	0.1
28	166	Sundar	Ande	6400	0.1
29	167	Amit	Banda	6200	0.1
30	179	Charles	Johnson	6200	0.1
31	177	Jack	Livingston	8400	0.2

32	176	Jonathon	Taylor	8600	0.2
33	175	Alyssa	Hutton	8800	0.25
34	174	Ellen	Abel	11000	0.3

The ROWID column is an internal identifier used by most relational DBMS products to identify each row of data, although the exact name of the column varies from one product to another. As you saw in Topic 5.2, ROWID values are also essential for indexing because they tie a data value in the index to the row(s) of data that contain matching data values in one or more columns (much like an index in the back of book uses page numbers to link to the text where a word or phrase appears in the book). If we take the concept a bit further, the table can be split into individual columns, with the columns logically tied together using the ROWID. Here are the Employee ID and First Name columns as they might be stored by a columnar DBMS.

Employee ID

ROWID	EMPLOYEE_ID
10	145
12	146
11	147
16	148
9	149
7	150
8	151
13	152
15	153
17	154
14	155

Employee ID

ROWID	EMPLOYEE_ID
18	156
23	157
22	158
21	159
20	160
19	161
24	162
25	163
26	164
27	165
28	166
29	167
6	168
5	169
4	170
3	171
2	172
1	173
34	174
33	175

Employee ID

ROWID	EMPLOYEE_ID
32	176
31	177
30	179

Employee First Name

ROWID	FIRST_NAME
11	Alberto
22	Allan
33	Alyssa
29	Amit
30	Charles
15	Christopher
24	Clara
25	Danielle
8	David
27	David
9	Eleni
2	Elizabeth
34	Ellen
16	Gerald

5	Harrison
31	Jack
18	Janette
10	John
32	Jonathon
12	Karen
21	Lindsey
6	Lisa
20	Louise
26	Mattea
17	Nanette
14	Oliver
23	Patrick
7	Peter
13	Peter
19	Sarath
28	Sundar
1	Sundita
4	Tayler
3	William

With row-oriented databases, the DBMS reads rows of data, and scans rows to find the columns selected in a query. With columnar databases, the DBMS only needs to read the selected columns, which speeds up processing considerably, particularly for tables with many columns.

You may have noticed that the columns shown in the illustration are sorted by the data values (Employee ID and First Name) instead of by the ROWID. Using this technique, the columnar database doesn't need indexes because the columns sorted by the data values serve the same purpose.

Considerations during the physical design of columnar databases include the following:

- **Column Grouping**: which columns to store together (as groups of columns). Columns that are usually used together, such a first and last name, can be more efficiently processed if stored together. However, you then must choose which column is used for ordering, and you may lose other optimization options.
- **Column Ordering**: which columns are maintained in data sequence so they can function as indexes. The tradeoff, of course, is the overhead of keeping the data in order as rows are inserted, updated, and deleted.
- **Column Compression**: Single columns are easier to compress than entire rows, so compression works better in columnar databases compared to relational databases. For example, columns that contain repeating values can be compressed by storing the repeating value once, followed by how many repetitions of the value the DBMS removed during compression. Another technique for columns with uniform intervals of values (such as identifiers that usually increase by 1 for each new logical row of data), is to store the first value along with the interval that is to be repeated until another value appears. For example, if the value 100 and the increment +1 is stored, the DBMS knows that the values are 100, 101, 102, and so forth, until the next value stops the sequence.
- **Distribution**: how the column data is to be distributed across the nodes in the cluster. A common method is to hash the column values. (Recall that hashing maps data of arbitrary sizes to values of a fixed size, typically in the form of binary or integer values). The hashed values can then be mapped uniformly across the nodes in the cluster. Another common distribution method is to assign ranges of values to specific nodes. However, this method becomes problematic when nodes are added or removed. It is also possible to replicate data across multiple nodes, particularly for columns used to join values from one logical table to another logical table. The key to join performance in clustered databases is to place values that will be joined on the same node so that data does not have to be transmitted (broadcast) from one node to another in order to complete the join operation. For example, if orders are commonly joined with customer accounts, then the Customer ID column from orders (the foreign keys) and the Customer ID column from customer accounts (the primary keys) should be distributed in the same way.

Document Databases

MongoDB is, by many measures, the most popular document database, so I use it here as a representative technology. A MongoDB instance can manage multiple databases and each MongoDB database can contain many collections. You can think of a MongoDB database as the equivalent of a relational schema. A MongoDB collection is similar to a relational table in that the documents in a collection describe different instances of the same entity. When you store a document in MongoDB, you must specify the database and the collection to which the document belongs.

One of the striking differences with document databases is that the structure of documents can vary within the same collection. In other words, one document in a collection can have data elements (fields) that do not exist in other documents in the same collection. This is not quite the same as NULL values in a relational database because NULL values indicate columns for which the value is unknown, whereas documents with missing data elements carry no assumed meaning – they could be missing because their values are unknown, or because they do not apply to that particular occurrence of the entity, or perhaps because the document is an older version that hasn't been updated to include the newer data elements.

MongoDB supports replicating data by defining replica sets, typically with each replica set going to a different node in the cluster, with one replica (copy) in each set designated as the master. Write consistency is then controlled by specifying the number of replicas in a set that must be written before a write operation is considered complete. At one extreme, if you choose consistency across all slaves, then if one node containing a replica with the set is offline, none of the write operations can complete. At the other extreme, if you choose just one replica per set for consistency, and the node containing that replica set is lost, you end up on the verge of losing data because all that remains is the master. As always, selecting the right number of replica sets for write consistency is a trade-off, and most designers choose middle ground close to a simple majority of the replica sets.

For larger databases, documents can be distributed (aplit) across cluster nodes by specifying a field within the document to be used for determining the distribution. In MongoDB, the term used for distribution is known as ***sharding***.

Unlike relational databases, MongoDB does not support the concept of a transaction that can span multiple documents such as a series of insert, update and delete operations across multiple documents. The only operations that can be atomic (completely successful or having no effect) are those that are applied to the same document.

MongoDB has a query language that is expressed via JSON using constructs to specify filtering, ordering, and specifying an execution plan. For example, a query to select the firstName and

lastName for the employee with the employeeID of 147 would look like this in SQL (using camelCase names):

SELECT firstName, lastName FROM employee WHERE employeeId = '147';

And like this in MongoDB:

db.employee.find({employeeId:"147"},{firstName:1,lastName:1})

As you can see, the syntax is very, very different.

Key-value Databases

Key-value store databases have the simplest physical structures of all the NoSQL databases. The database user has only three possible operations: get (read) the value for a key, put (write) the value for key, or delete the key from the data store. The value is simply a blob (binary large object) that is stored alongside the key. This simplicity lends itself well to databases stored entirely in memory as well as disk systems.

The concept of consistency applies only to operations on a single key because each key-value pair is stored separately from all others. Said another way, the database does not support relationships between different keys, so each key-value pair is independent of all others. For distributed key-value databases, such as Riak, replication of key-value pairs is handled asynchronously. Therefore, at any point in time, some nodes in the cluster will be inconsistent with other nodes, but if no more puts or deletes are processed, the replicated copies on all nodes will eventually become consistent. This consistency model if called ***eventual consistency***.

Support for transactions varies quite a bit from one product to another, but in general, there are no guarantees on writes because of the eventual consistency model. However, some products offer support for a minimum number of clusters that must be written to before a write is considered complete. This is similar to the replica set supported by MongoDB. Riak, for example, uses the concept of quorums, where a write is considered complete when a put has been completed on a majority of the applicable nodes

Queries are generally limited to searching (getting) values using their corresponding keys. Generally, you cannot search based on the contents of the values, but only on the keys. However, a few products do allow searching on the values. For example, Riak includes a feature known as Riak Search that permits searching based on characteristics of the values stored.

For scaling large amounts of data, most key-value stores support the concept of sharding, with the key being used to determine the node on which the key-value pair will be stored. For example, the

first character of the key might be used to determine the storage node. One downside of sharding occurs when a node is unavailable. For example, if the node used to store keys starting with "e" goes down, we cannot retrieve any values for keys starting with "e", nor can we write any values that have keys starting with "e".

Graph Databases

As you saw in Topic 1.4, graph databases are radically different from other databases in that they focus more on storing the relationships (edges) between records of values (nodes; not to be confused with cluster nodes) than they do on storing the values themselves. This leads to some interesting storage considerations, including the following:

- Graph databases usually store only one type of relationship, such as "which condition was this medication prescribed to treat?".
- Adding additional relationships to an existing graph database usually leads to a lot of schema changes and movement of data.
- Relationships between nodes are created in both directions because relationships are not necessarily bi-directional. For example, a medication might be prescribed to treat a medical condition, but the medical condition might not indicate the medication as the recommended treatment for the condition.

For consistency, most graph databases do not support distributing (sharding) nodes across different servers in the cluster. However, some support replication with updates to slave copies following the eventual consistency model.

Some graph databases support transactions. For example, Neo4J provides full transaction support. Before any changes to nodes or relationships in Neo4J, a transaction must be started. Transactions are completed by marking them successful, followed by marking them completed by invoking a finish method. Data changes are not committed until the finish is issued. If a transaction fails, it is rolled back to the beginning, much like relational database transactions. Reads, however, can be performed completely outside of transactions.

Neo4J uses the master-slave paradigm (a master and one or more replicated slaves) to support availability. Writes are committed to the master first, then one slave. Additional slaves follow the eventual consistency model. Other graph databases, such as FlockDB and Infinite Graph, support distributing nodes across the servers in the cluster.

For querying, Neo4J includes the Cypher query language. Neo4J also supports language bindings for querying a graph to find properties of nodes, traversing a graph, and navigating relationships associated with nodes. Gemini is a language specifically designed for traversing graphs, and is

capable of traversing all databases that implement the Blueprints property graph, which includes not only Neo4J, but most other graph database products.

Graph databases have to be approached differently when we need to scale them to handle very large graphs. The sharding techniques used on most other NoSQL databases does not work well with graph databases because each node can potentially be connected to many other nodes. Therefore, one or more of the following techniques must be applied in order to scale graph databases:

- Add sufficient memory so that active nodes and relationships will usually fit in memory. (Memory is considerably faster than disk for accessing data.)
- Add more slaves to allow slaves to share the load for reads, while restricting all write operations to the master.
- If sharding must be done, distribute the nodes based on subsets of the nodes that always tend to be used together. For example, for the medical condition and medication example mentioned earlier, we might be successful if we shard the nodes based on medical specialty (conditions and medications within cardiology on one server, ophthalmology on another, and so forth). This technique requires domain knowledge of the data as well as an understanding of how the graph database is typically used.