W

MATERIALIZED VIEWS

- A materialized view stored both definition of the view plus the rows resulting from the execution of the query with the view
- It is more efficient to use materialized views if the query involves summaries, large or multiple joins or both
- It is a pre computed table comprising aggregated or joined data from more than 1 table.
- It is also known as summary or aggregate table and is mainly used for improving query performance or providing replicated data

Types of materialized views

- Materialized view with aggregates
- Materialized views containing joins

MATERIALIZED VIEW LOGS

- They are used when we want to use fast refresh
- Materialized view logs are defined using create materialized view log statement on the base table that is to be changed
- For fast refresh of materialized views the definition of materialized view logs must normally specify the rowed clause.
- In addition for aggregate materialized views, it must also contain every column in the table referenced in the materialized view, the including new values clause and the sequence clause

Materialized view refresh modes

Manual refresh :- can be performed using dbms_mview package

Automatic Rfefresh :- can be performed in 2 ways

(a) On Commit – Materialized views get updated when ever changes to one of the tables are committed.

(b) On demand – At specified time refresh is scheduled to occur for specified time.

```
create materialized view log on dept

with rowid(deptno,dname,loc) including new values;

create materialized view log on emp

with rowid(empno,ename,job,mgr,hiredate,sal,comm,deptno) including new values;

create materialized view fmv2
refresh fast on commit
enable query rewrite
as
select empno,ename,sal,dname,loc,e.rowid er,d.rowid dr
from emp e,dept d
where e.deptno = d.deptno
```

SQL vs NoSQL: High-Level Differences

- SQL databases are primarily called as Relational Databases (RDBMS); whereas NoSQL database are primarily called as non-relational or distributed database.
- SQL databases are table based databases whereas NoSQL databases are document based, key-value pairs, graph databases or wide-column stores. This means that SQL databases represent data in form of tables which consists of n number of rows of data whereas NoSQL databases are the collection of key-value pair, documents, graph databases or wide-column stores which do not have standard schema definitions which it needs to adhered to.
- SQL databases have predefined schema whereas NoSQL databases have dynamic schema for unstructured data.
- SQL databases are vertically scalable whereas the NoSQL databases are horizontally scalable. SQL databases are scaled by increasing the horse-power of the hardware. NoSQL databases are scaled by increasing the databases servers in the pool of resources to reduce the load.
- SQL databases uses SQL (structured query language) for defining and manipulating the data, which is very powerful. In NoSQL database, queries are focused on collection of documents. Sometimes it is also called as UnQL (Unstructured Query Language). The syntax of using UnQL varies from database to database.
- SQL database examples: MySql, Oracle, Sqlite, Postgres and MS-SQL. NoSQL database examples: MongoDB, BigTable, Redis, RavenDb, Cassandra, Hbase, Neo4j and CouchDb
- For complex queries: SQL databases are good fit for the complex query intensive environment whereas NoSQL databases are not good fit for complex queries. On a high-level, NoSQL don't have standard interfaces to perform complex queries, and the queries themselves in NoSQL are not as powerful as SQL query language.
- For the type of data to be stored: SQL databases are not best fit for hierarchical data storage. But, NoSQL database fits better for the hierarchical data storage as it follows the key-value pair way of storing data similar to JSON data. NoSQL database are highly preferred for large data set (i.e for big data). Hbase is an example for this purpose.
- For scalability: In most typical situations, SQL databases are vertically scalable. You can manage increasing load by increasing the CPU, RAM, SSD, etc, on a single server. On the other hand, NoSQL databases are horizontally scalable. You can just add few more servers easily in your NoSQL database infrastructure to handle the large traffic.
- For high transactional based application: SQL databases are best fit for heavy duty transactional type applications, as it is more stable and promises the atomicity as well as integrity of the data. While you can use NoSQL for transactions purpose, it is still not comparable and sable enough in high load and for complex transactional applications.
- For support: Excellent support are available for all SQL database from their vendors. There are also lot of independent consultations who can help you with SQL database for a very large scale deployments. For some NoSQL database you still have to rely on community support, and only limited outside experts are available for you to setup and deploy your large scale NoSQL deployments.

- For properties: SQL databases emphasizes on ACID properties (Atomicity, Consistency, Isolation and Durability) whereas the NoSQL database follows the Brewers CAP theorem (Consistency, Availability and Partition tolerance)
- For DB types: On a high-level, we can classify SQL databases as either open-source or close-sourced from commercial vendors. NoSQL databases can be classified on the basis of way of storing data as graph databases, key-value store databases, document store databases, column store database and XML databases.

What is NoSQL?

NoSQL encompasses a wide variety of different database technologies and were developed in response to a rise in the volume of data stored about users, objects and products, the frequency in which this data is accessed, and performance and processing needs. Relational databases, on the other hand, were not designed to cope with the scale and agility challenges that face modern applications, nor were they built to take advantage of the cheap storage and processing power available today.

NoSQL Database Types

- **Document databases** pair each key with a complex data structure known as a document. Documents can contain many different key-value pairs, or key-array pairs, or even nested documents.

- **Graph stores** are used to store information about networks, such as social connections. Graph stores include Neo4J and HyperGraphDB.

- **Key-value stores** are the simplest NoSQL databases. Every single item in the database is stored as an attribute name (or "key"), together with its value. Examples of key-value stores are Riak and Voldemort. Some key-value stores, such as Redis, allow each value to have a type, such as "integer", which adds functionality.

- **Wide-column stores** such as Cassandra and HBase are optimized for queries over large datasets, and store columns of data together, instead of rows.

The Benefits of NoSQL

When compared to relational databases, NoSQL databases are more scalable and provide superior performance, and their data model addresses several issues that the relational model is not designed to address:

- Large volumes of structured, semi-structured, and unstructured data
- Agile sprints, quick iteration, and frequent code pushes
- Object-oriented programming that is easy to use and flexible
- Efficient, scale-out architecture instead of expensive, monolithic architecture

Dynamic Schemas

Relational databases require that schemas be defined before you can add data. For example, you might want to store data about your customers such as phone numbers, first and last name, address, city and state – a SQL database needs to know what you are storing in advance.

This fits poorly with agile development approaches, because each time you complete new features, the schema of your database often needs to change. So if you decide, a few iterations into development, that you'd like to store customers' favorite items in addition to their addresses and phone numbers, you'll need to add that column to the database, and then migrate the entire database to the new schema.

If the database is large, this is a very slow process that involves significant downtime. If you are frequently changing the data your application stores – because you are iterating rapidly – this downtime may also be frequent. There's also no way, using a relational database, to effectively address data that's completely unstructured or unknown in advance.

NoSQL databases are built to allow the insertion of data without a predefined schema. That makes it easy to make significant application changes in real-time, without worrying about service interruptions – which means development is faster, code integration is more reliable, and less database administrator time is needed.

Auto-sharding

Because of the way they are structured, relational databases usually scale vertically – a single server has to host the entire database to ensure reliability and continuous availability of data. This gets expensive quickly, places limits on scale, and creates a relatively small number of failure points for database infrastructure. The solution is to scale horizontally, by adding servers instead of concentrating more capacity in a single server.

"Sharding" a database across many server instances can be achieved with SQL databases, but usually is accomplished through SANs and other complex arrangements for making hardware act as a single server. Because the database does not provide this ability natively, development teams take on the work of deploying multiple relational databases across a number of machines. Data is stored in each database instance autonomously. Application code is developed to distribute the data, distribute queries, and aggregate the results of data across all of the database instances. Additional code must be developed to handle resource failures, to perform joins across the different databases, for data rebalancing, replication, and other requirements. Furthermore, many benefits of the relational database, such as transactional integrity, are compromised or eliminated when employing manual sharding.

NoSQL databases, on the other hand, usually support auto-sharding, meaning that they natively and automatically spread data across an arbitrary number of servers, without requiring the application to even be aware of the composition of the server pool. Data and query load are automatically balanced across servers, and when a server goes down, it can be quickly and transparently replaced with no application disruption.

Cloud computing makes this significantly easier, with providers such as Amazon Web Services providing virtually unlimited capacity on demand, and taking care of all the necessary database administration tasks. Developers no longer need to construct complex, expensive platforms to support their applications, and can concentrate on writing application code. Commodity servers can provide the same processing and storage capabilities as a single high-end server for a fraction of the price.

Replication

Most NoSQL databases also support automatic replication, meaning that you get high availability and disaster recovery without involving separate applications to manage these tasks. The storage environment is essentially virtualized from the developer's perspective.

Integrated Caching

A number of products provide a caching tier for SQL database systems. These systems can improve read performance substantially, but they do not improve write performance, and they add complexity to system deployments. If your application is dominated by reads then a distributed cache should probably be considered, but if your application is dominated by writes or if you have a relatively even mix of reads and writes, then a distributed cache may not improve the overall experience of your end users.

Many NoSQL database technologies have excellent integrated caching capabilities, keeping frequently-used data in system memory as much as possible and removing the need for a separate caching layer that must be maintained.

NoSQL vs. SQL Summary

	SQL Databases	NoSQL Databases
Types	One type (SQL database) with minor variations	Many different types including key-value stores, document databases, wide-column stores, and graph databases
Development History	Developed in 1970s to deal with first wave of data storage applications	Developed in 2000s to deal with limitations of SQL databases, particularly concerning scale, replication and unstructured data storage
Examples	MySQL, Postgres, Oracle Database	MongoDB, Cassandra, HBase, Neo4j
Data Storage Model	Individual records (e.g., "employees") are stored as rows in tables, with each column storing a specific piece of data about that record (e.g., "manager," "date hired," etc.), much like a spreadsheet. Separate data types are stored in separate tables, and then joined together when more complex queries are executed. For example, "offices" might be stored in one table, and "employees" in another. When a user wants to find the work address of an employee, the database engine joins the "employee" and "office" tables together to get all the information necessary.	Varies based on database type. For example, key-value stores function similarly to SQL databases, but have only two columns ("key" and "value"), with more complex information sometimes stored within the "value" columns. Document databases do away with the table-and-row model altogether, storing all relevant data together in single "document" in JSON, XML, or another format, which can nest values hierarchically.
Schemas	Structure and data types are fixed in advance. To store information about a new data item, the entire database must be altered, during which time the database must be taken offline.	Typically dynamic. Records can add new information on the fly, and unlike SQL table rows, dissimilar data can be stored together as necessary. For some databases (e.g., wide-column stores), it is somewhat more challenging to add new fields dynamically.
Scaling	Vertically, meaning a single server must be made increasingly powerful in order to deal with increased demand. It is possible to spread SQL databases	Horizontally, meaning that to add capacity, a database administrator can simply add more commodity servers or cloud instances. The database

	SQL Databases	NoSQL Databases
	over many servers, but significant additional engineering is generally required.	automatically spreads data across servers as necessary
Development Model	Mix of open-source (e.g., Postgres, MySQL) and closed source (e.g., Oracle Database)	Open-source
Supports Transactions	Yes, updates can be configured to complete entirely or not at all	In certain circumstances and at certain levels (e.g., document level vs. database level)
Data Manipulation	Specific language using Select, Insert, and Update statements, e.g. SELECT fields FROM table WHERE…	Through object-oriented APIs
Consistency	Can be configured for strong consistency	Depends on product. Some provide strong consistency (e.g., MongoDB) whereas others offer eventual consistency (e.g., Cassandra)

Oracle Table Clustering is an optional technique of storing table data. The primary purpose of Clustering is to take the rows from 1 or more tables that are associated to each other about a common data entity, and physically store those rows adjacent to each other in the same disk block This common data entity is defined as the 'Cluster Key'. Idenfifying the Cluster Key is the first step to implement Oracle Table Clustering.

The advantage of clustering is twofold. Firstly, by storing the field or fields comprising the Cluster Key once instead of multiple times, storage is saved. More significantly, the fields of the Cluster Key are often indexed, so the high maintenance indexes for those fields are also reduced from containing multiple key entries to a single key entry.

The arguably more significant advantage to Clustering is to expidite join queries. When 2 (or more) tables repeatedly have their rows accessed together via a join query, those tables are good candidates for Clustering. Since the rows of both tables would be stored adjacently in the same disk block if they were clustered, when a query is done that joins these 2 tables about the Cluster Key, the joined rows would be fetched with a single IO operation (see lists of I/O operations).

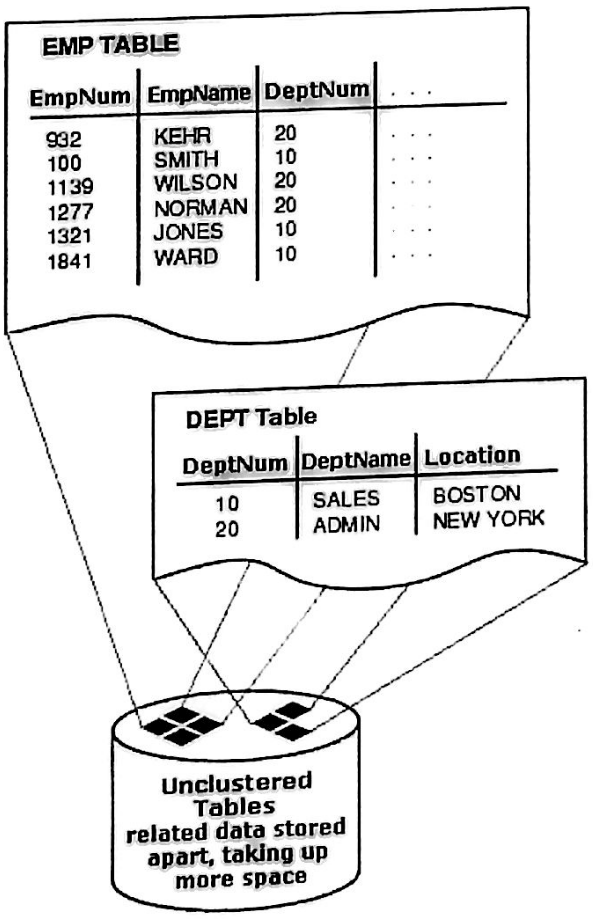

figure 1

CREATE CLUSTER

Use the CREATE CLUSTER statement to create a cluster. A **cluster** is a schema object that contains data from one or more tables, all of which have one or more columns in common. Oracle Database stores together all the rows from all the tables that share the same cluster key.

For information on existing clusters, query the USER_CLUSTERS, ALL_CLUSTERS, and DBA_CLUSTERS data dictionary views

To create a cluster in your own schema, you must have CREATE CLUSTER system privilege. To create a cluster in another user's schema, you must have CREATE ANY CLUSTER system privilege. Also, the owner of the schema to contain the cluster must have either space quota on the tablespace containing the cluster or the UNLIMITED TABLESPACE system privilege.

Oracle Database does not automatically create an index for a cluster when the cluster is initially created. Data manipulation language (DML) statements cannot be issued against cluster tables in an indexed cluster until you create a cluster index with a CREATE INDEX statement.

The following statement creates a cluster named personnel with the cluster key column department, a cluster size of 512 bytes, and storage parameter values:

```
CREATE CLUSTER personnel
  (department NUMBER(4))
SIZE 512
STORAGE (initial 100K next 50K);
```

The following statement creates the cluster index on the cluster key of personnel:

```
CREATE INDEX idx_personnel ON CLUSTER personnel;
```

After creating the cluster index, you can add tables to the index and perform DML operations on those tables.

Adding Tables to a Cluster: Example

The following statements create some departmental tables from the sample hr.employees table and add them to the personnel cluster created in the earlier example:

```
CREATE TABLE dept_10
  CLUSTER personnel (department_id)
  AS SELECT * FROM employees WHERE department_id = 10;

CREATE TABLE dept_20
  CLUSTER personnel (department_id)
  AS SELECT * FROM employees WHERE department_id = 20;
```

A table cluster is a group of tables that share the same data blocks, since they share common columns and are often used together. When you create cluster tables, Oracle physically stores all rows for each table in the same data blocks. The cluster key value is the value of the cluster key columns for a particular row.