Welcome

Database Design and Optimization

Prakash Badhe

prakash.badhe@vishwasoft.in

Copyright Notice

Agenda

- Normalization to avoid redundant data tables
- Queries with Joins, Group by, Having
- Pivot and Unpivot Queries with Case expression
- Simple Queries with timeseries tables
- The name value pair tables pattern
- The audit tables vs. the audit via temporal Tables
- The Query plans and Indexing
- The Database partitions
- Optimizing the database design for performance

Database setup

- Mysql ver 5.1.30
- https://downloads.mysql.com/archives/community/
- WorkBench clients ver 6.1.7
- https://dev.mysql.com/downloads/workbench/
- MySQL JDBC Driver

Database normalization

- Database normalization is the process of structuring a relational database in accordance with a series of so-called normal forms in order to reduce data redundancy and improve data integrity.
- First proposed by Edgar F. Codd as part of the relational model.

Objectives of Normalization

- To permit data to be queried and manipulated using a "universal data sub-language" grounded in first-order logic.
- To free the collection of relations from undesirable insertion, update and deletion dependencies.
- To reduce the need for restructuring the collection of relations, as new types of data are introduced, and thus increase the life span of application programs.

Objectives..

- To make the relational model more informative to users.
- To make the collection of relations neutral to the query statistics, where these statistics are liable to change as time goes by.

Side effects of Modification

- When an attempt is made to modify (update, insert into, or delete from) a relation, the undesirable side-effects may arise in relations that have not been sufficiently normalized
- **Update anomaly.** The same information can be expressed on multiple rows; therefore updates to the relation may result in logical inconsistencies.

Update Side-Effect

- Each record in an "Employees ' Skills" relation contains an Employee ID, Employee Address, and Skill.
- Thus a change of address for a particular employee may need to be applied to multiple records (one for each skill).
- If the update is only partially successful – the employee's address is updated on some records but not others – then the relation is left in an inconsistent state.
- Specifically, the relation provides conflicting answers to the question of what this particular employee's address is.
- This phenomenon is known as an update anomaly.

Insertion side-effect

- **Insertion anomaly.** There are circumstances in which certain facts cannot be recorded at all.
- For example, each record in a "Faculty and Their Courses" relation might contain a Faculty ID, Faculty Name, Faculty Hire Date, and Course Code.
- Therefore, we can record the details of any faculty member who teaches at least one course, but we cannot record a newly hired faculty member who has not yet been assigned to teach any courses, except by setting the Course Code to null.
- This phenomenon is known as an insertion anomaly.

Delete side-effect

- **Deletion anomaly.**
- Under certain circumstances, deletion of data representing certain facts necessitates deletion of data representing completely different facts.
- The "Faculty and Their Courses" relation in the previous case suffers from this type of anomaly, for if a faculty member temporarily ceases to be assigned to any courses, we must delete the last of the records on which that faculty member appears, effectively also deleting the faculty member, unless we set the Course Code to null.
- This phenomenon is known as a deletion anomaly.

Normalization Goals

- **Minimize redesign when extending the database structure**
- A fully normalized database allows its structure to be extended to accommodate new types of data without changing existing structure too much.
- As a result, applications interacting with the database are minimally affected.
- Normalized relations, and the relationship between one normalized relation and another, mirror real-world concepts and their interrelationships.

Normalization Guidelines

- Codd introduced the concept of normalization and what is now known as the first normal form (1NF).
- Codd went on to define the second normal form (2NF) and third normal form (3NF).
- Codd and Raymond F. Boyce also defined the Boyce-Codd normal form (BCNF).
- A relational database relation is often described as "normalized" if it meets third normal form specification.
- Most 3NF relations are free of insertion, update, and deletion anomalies.

Constraints for Normalization

- Primary key (no duplicate tuples)
- Atomic columns (cells have single value)
- No partial dependencies (values depend on the whole of every Candidate key)
- No transitive dependencies (values depend only on Candidate keys)
- No redundancy from any functional dependency

Constraints for Normalization..

- The component of every explicit join dependency is a superkey.
- Every non-trivial join dependency is implied by a candidate key.
- Every constraint is a consequence of domain constraints and key constraints
- Every join dependency is trivial.

Normalization step by step

- Normalization is a design technique used to design a relational database table up to higher normal form.
- The process is progressive, and a higher level of database normalization cannot be achieved unless the previous levels have been satisfied.
  - Separate the duplicities into multiple columns using repeating groups
  - Identify entities represented in the table and separate them into their own respective tables.
  - The relationships between the newly introduced tables need to be determined.

SQL Join

- A SQL Join statement is used to combine data or rows from two or more tables based on a common field between them.
- A JOIN is a means for combining fields from two tables by using values common to each.

Join Example

- CUSTOMERS Table Columns
- ID | NAME | AGE | ADDRESS | SALARY
- ORDERS Table columns
- OID | DATE | CUSTOMER_ID | AMOUNT
- Join these two tables in SELECT statement as
- SELECT ID, NAME, AGE, AMOUNT FROM CUSTOMERS, ORDERS WHERE CUSTOMERS.ID = ORDERS.CUSTOMER_ID;

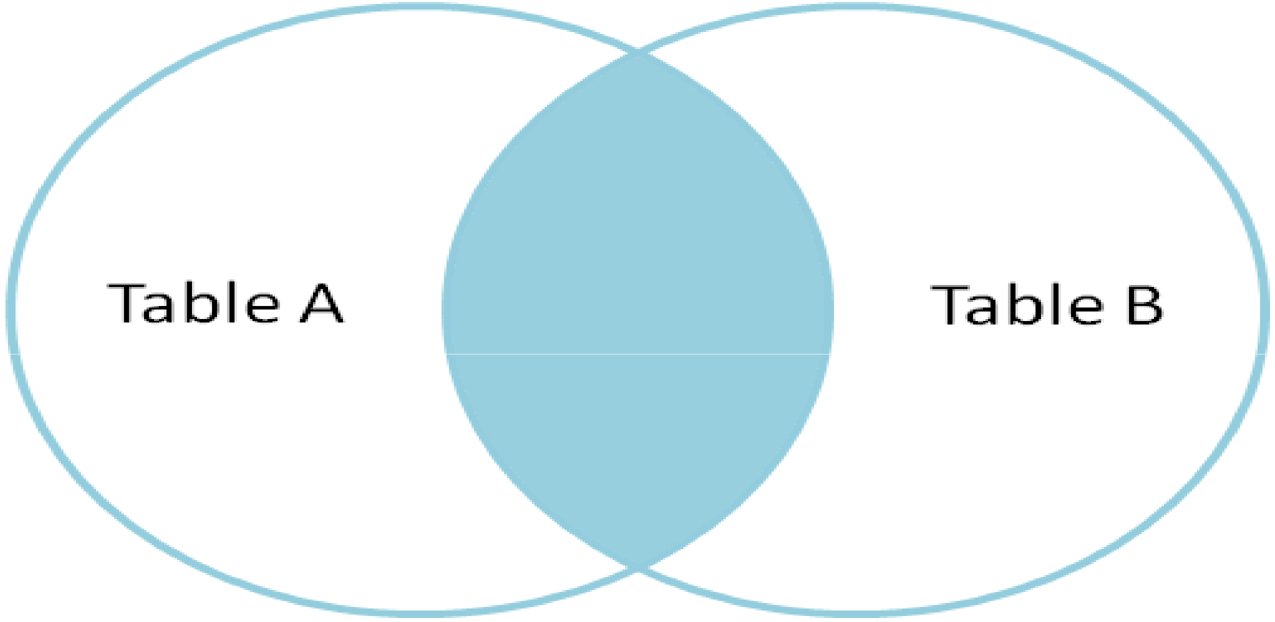
Join Operator

- The join is performed in the WHERE clause.
- Several operators can be used to join tables, such as =, <, >, <>, <=, >=, !=, BETWEEN, LIKE, and NOT;
- They can all be used to join tables.
- However, the most common operator is the equal to symbol.

Join Types

- **INNER JOIN**
- **LEFT JOIN**
- **RIGHT JOIN**
- **FULL JOIN**
- **CARTESIAN JOIN**
- **SELF JOIN**

Inner Join of Two Tables



Show the names and age of students enrolled in different courses.

SELECT StudentCourse.COURSE_ID, Student.NAME, Student.AGE FROM Student INNER JOIN StudentCourse ON Student.ROLL_NO = StudentCourse.ROLL_NO;

INNER JOIN

- The INNER JOIN keyword selects all rows from both the tables as long as the condition satisfies.
- This keyword will create the result-set by combining all rows from both the tables where the condition satisfies i.e value of the common field will be same.

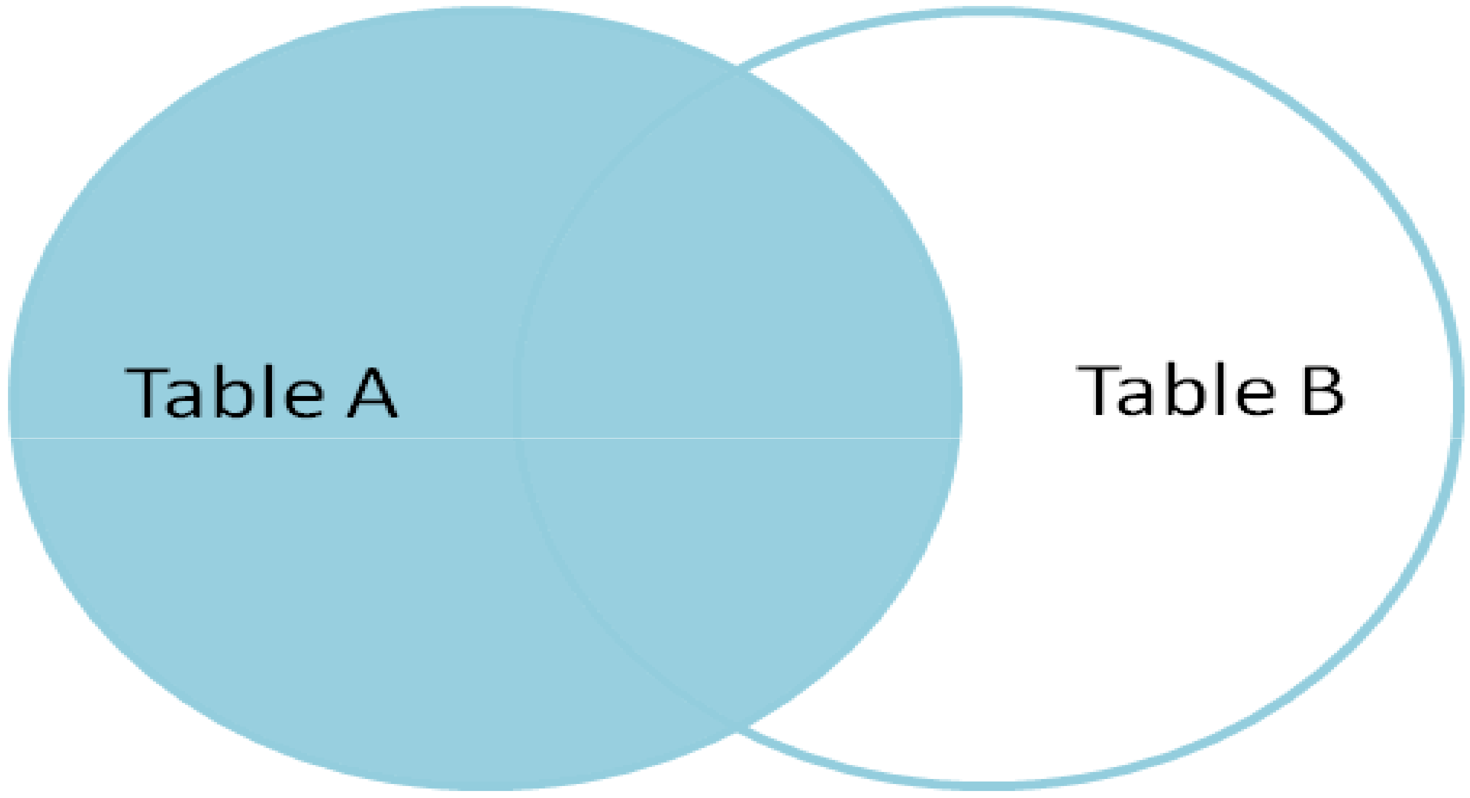
Inner join Syntax

- SELECT table1.column1,table1.column2,table2.column1,.... FROM table1 INNER JOIN table2 ON table1.matching_column = table2.matching_column;
- **table1**: First table.
- **table2**: Second table
- **matching_column**: Column common to both the tables.
  - Instead of INNER JOIN only JOIN can be used. JOIN is same as INNER JOIN.

LEFT JOIN

- This join returns all the rows of the table on the left side of the join and matching rows for the table on the right side of join.
- The rows for which there is no matching row on right side, the result-set will contain *null*.
- LEFT JOIN is also known as LEFT OUTER JOIN

Left Outer Join



```
SELECT Student.NAME,StudentCourse.COURSE_ID FROM Student
LEFT JOIN StudentCourse ON StudentCourse.ROLL_NO =
Student.ROLL_NO;
```

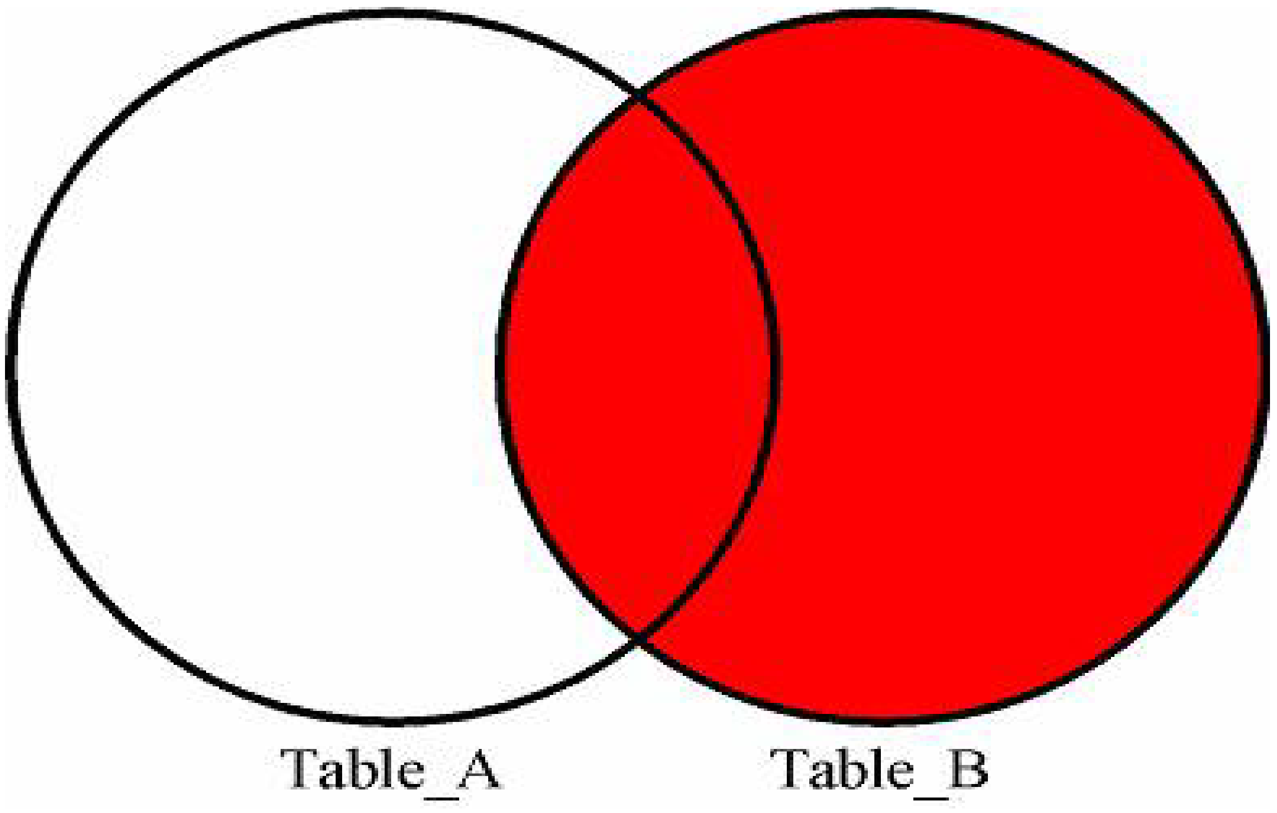
Left Join Syntax

- SELECT table1.column1,table1.column2,table2.column1,.... FROM table1 LEFT JOIN table2 ON table1.matching_column = table2.matching_column;
- table1: First table.
- table2: Second table
- matching_column: Column common to both the tables.

Right Join

- RIGHT JOIN is similar to LEFT JOIN.
- This join returns all the rows of the table on the right side of the join and matching rows for the table on the left side of join.
- The rows for which there is no matching row on left side, the result-set will contain *null*.
- RIGHT JOIN is also known as RIGHT OUTER JOIN.

Right Outer Join

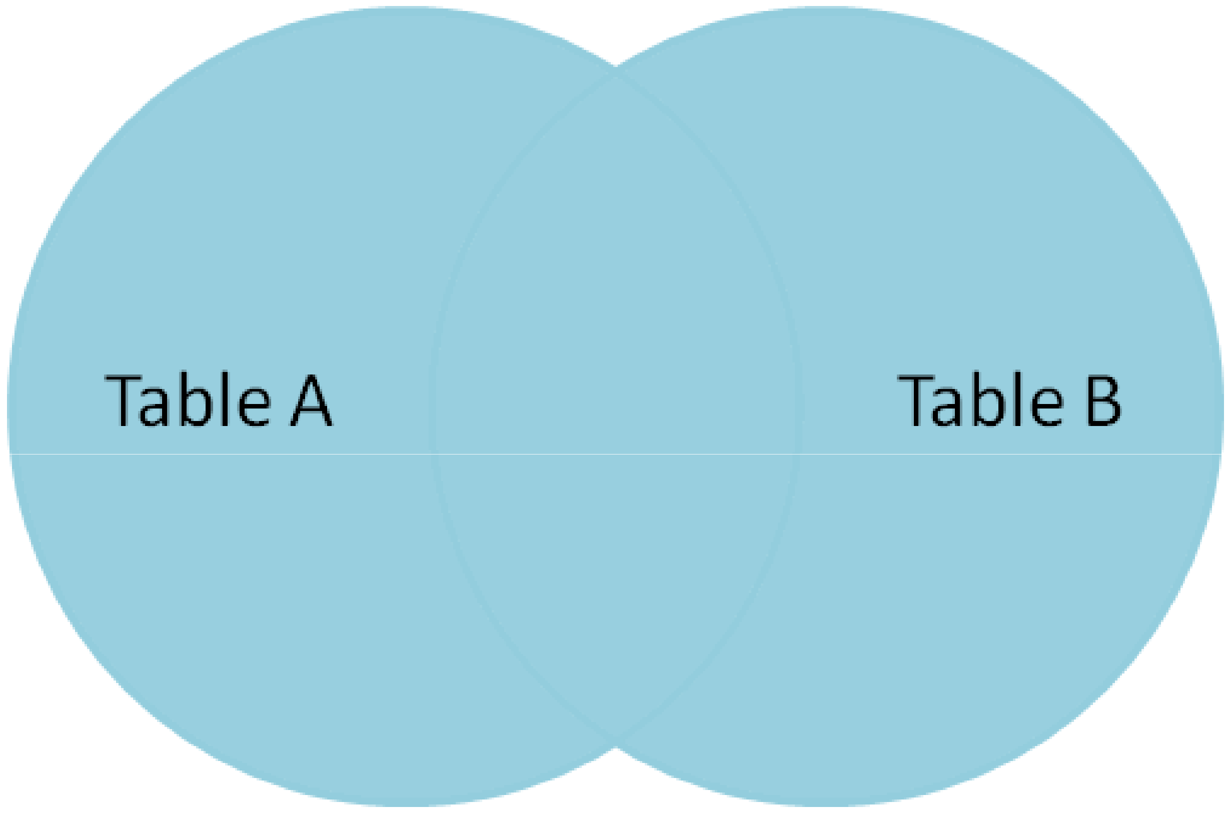


```
SELECT Student.NAME,StudentCourse.COURSE_ID
FROM Student RIGHT JOIN StudentCourse ON
StudentCourse.ROLL_NO = Student.ROLL_NO;
```

Full Join

- FULL JOIN creates the result-set by combining result of both LEFT JOIN and RIGHT JOIN.
- The result-set will contain all the rows from both the tables. The rows for which there is no matching, the result-set will contain *NULL* values.

Full Join



```
SELECT Student.NAME,StudentCourse.COURSE_ID FROM
Student FULL JOIN StudentCourse ON
StudentCourse.ROLL_NO = Student.ROLL_NO;
```

Cartesian Join

- The CARTESIAN JOIN is also known as CROSS JOIN.
- In a CARTESIAN JOIN there is a join for each row of one table to every row of another table.
- This usually happens when the matching column or WHERE condition is not specified.
- In the absence of a WHERE condition the CARTESIAN JOIN will behave like a CARTESIAN PRODUCT . i.e., the number of rows in the result-set is the product of the number of rows of the two tables.
- In the presence of WHERE condition this JOIN will function like a INNER JOIN.
- The Cross join is similar to an inner join where the join-condition will always evaluate to True.

CARTESIAN JOIN Syntax

SELECT table1.column1 , table1.column2, table2.column1... FROM table1 CROSS JOIN table2; **table1**: First table. **table2**: Second table

CARTESIAN JOIN Query

- Select NAME and Age from Student table and COURSE_ID from StudentCourse table.
- In the output each row of the table Student is joined with every row of the table StudentCourse.
- The total rows in the result-set = 4 * 4 = 16.
- SELECT Student.NAME, Student.AGE, StudentCourse.COURSE_ID FROM Student CROSS JOIN StudentCourse;

Self Join

- As the name signifies, in SELF JOIN a table is joined to itself.
- That is, each row of the table is joined with itself and all other rows depending on some conditions.
- In other words it is a join between two copies of the same table.
  - SELECT a.coulmn1 , b.column2 FROM table_name a, table_name b WHERE some_condition;
  - **table_name**: Name of the table.
  - **some_condition**: Condition for selecting the rows.

Self Join Query

```
SELECT a.ROLL_NO , b.NAME FROM Student a,
Student b WHERE a.ROLL_NO < b.ROLL_NO;
```

Join Query use case

- Combine aggregate and non-aggregate values in SQL using Joins and Over clause.
- Aggregate functions perform a calculation on a set of values and return a single value.
- Consider an employee table EMP and a department table DEPT.
- DISPLAY NAME, SAL, JOB OF EMP ALONG WITH MAX, MIN, AVG, TOTAL SAL OF THE EMPS DOING THE SAME JOB.
- DISPLAY DEPTNAME WITH NUMBER OF EMP WORKING IN IT.

Join Usage

- The aggregated values can’t be directly used with non-aggregated values to obtain a result.
- **Using Joins –**Create a sub-table containing the result of aggregated values.
- Using Join, use the results from the sub-table to display them with non-aggregated values

Join Query part1

```
SELECT ENAME, SAL, EMP.JOB,
   SUBTABLE.MAXSAL, SUBTABLE.MINSAL,
   SUBTABLE.AVGSAL, SUBTABLE.SUMSAL FROM
   EMP INNER JOIN (SELECT JOB, MAX(SAL)
   MAXSAL, MIN(SAL) MINSAL, AVG(SAL)
   AVGSAL, SUM(SAL) SUMSAL FROM EMP
   GROUP BY JOB) SUBTABLE ON EMP.JOB =
   SUBTABLE.JOB;
```

Over Clause

- **Using 'Over' clause –**OVER CLAUSE ALONG WITH PARTION BY IS USED TO BRAKE UP DATA INTO PARTITIONS.
- THE SPECIFIED FUNCTION OPERATES FOR EACH PARTITION.
- SELECT DISTINCT(DNAME), COUNT(ENAME) OVER (PARTITION BY EMP.DEPTNO) EMP FROM EMP RIGHT OUTER JOIN DEPT ON EMP.DEPTNO=DEPT.DEPTNO ORDER BY EMP DESC;

Having vs. Where Clause

- The difference between the having and where clause in SQL is that the where clause cann*ot* be used with aggregates, but the having clause can.
- The **where** clause works on row’s data, not on aggregated data.

Having in Query

- SELECT Student, Score FROM Marks WHERE Score >=40
- This would select data row by row basis.
- The **having** clause works on aggregated data.
- SELECT Student, SUM(score) AS total FROM Marks GROUP BY Student
- With having in above query, we get
- SELECT Student, SUM(score) AS total FROM Marks GROUP BY Student HAVING total > 70

SQL Group By

- The GROUP BY Statement in SQL is used to arrange identical data into groups with the help of some functions.
- Example : if a particular column has same values in different rows then it will arrange these rows in a group.
  - GROUP BY clause is used with the SELECT statement.
  - In the query, GROUP BY clause is placed after the WHERE clause.
  - In the query, GROUP BY clause is placed before ORDER BY clause if used any.

Group by Syntax

SELECT column1, function_name(column2) FROM table_name WHERE condition GROUP BY column1, column2 ORDER BY column1, column2;

function_name: Name of the function used

e.g. SUM() , AVG().

table_name: Name of the table.

condition: Condition used.

Group By single column

- Group By single column means, to place all the rows with same value of only that particular column in one group.
- Consider the query as :
- SELECT NAME, SUM(SALARY) FROM Employee GROUP BY NAME;

Group By multiple columns

- Group by multiple column
- **GROUP BY column1, column2**.
- This means to place all the rows with same values of both the columns **column1** and **column2** in one group.
- Query:SELECT SUBJECT, YEAR, Count(*) FROM Student GROUP BY SUBJECT, YEAR;
- The students with both same SUBJECT and YEAR are placed in same group. And those whose only SUBJECT is same but not YEAR belongs to different groups. So here we have grouped the table according to two columns or more than one column.

Having Clause

- WHERE clause is used to place conditions on columns but what if we want to place conditions on groups
- Use HAVING clause to place conditions to decide which group will be the part of final result-set. Also we can not use the aggregate functions like SUM(), COUNT() etc. with WHERE clause.
- So use HAVING clause to use any of these functions in the conditions.

Having Syntax

- SELECT column1, function_name(column2) FROM table_name WHERE condition GROUP BY column1, column2 HAVING condition ORDER BY column1, column2;
- **function_name**: Name of the function used for example, SUM() , AVG().
- **table_name**: Name of the table.
- **condition**: Condition used.
- SELECT NAME, SUM(SALARY) FROM Employee GROUP BY NAME HAVING SUM(SALARY)>3000;

Having usage

- Only one group out of the three groups appears in the result-set as it is the only group where sum of SALARY is greater than 3000.
- So we have used HAVING clause here to place this condition as the condition is required to be placed on groups not columns.

Where and Group By Clause usage

- **Where** and **Group By** clauses are used to filter rows returned by the query based on the condition.
- **WHERE clause** specifies search conditions for the rows returned by the Query and limits rows to a specific row-set.
- If a table has huge amount of records and if someone wants to get the particular records then using ‘where’ clause is useful.

Group By

- **GROUP BY clause** summaries identical rows into a single/distinct group and returns a single row with the summary for each group
- By using appropriate Aggregate function in the SELECT list, like COUNT(), SUM(), MIN(), MAX(), AVG(), etc.

Where Usage

Get a list of Customers who bought some number of items last year, so that they can sell more some stuff to them this year.

With table called SalesOrder with columns CustomerId, SalesOrderId, Order_Date, OrderNumber, OrderItem, UnitPrice, OrderQty - need to get the customers who made orders last year i.e. 2017

Where and Group By

- **Using Where clause –**SELECT * FROM [Sales].[Orders] WHERE Order_Date >= '2017-01-01 00:00:00.000' AND Order_Date < '2018-01-01 00:00:00.000'
- Return the row set with all the Customers and corresponding Orders of year 2017.
- **Using Group By clause –**
- SELECT CustomerID, COUNT(*) AS OrderNumbers FROM [Sales].[Orders] WHERE Order_Date >= '2017-01-01 00:00:00.000' AND Order_Date < '2018-01-01 00:00:00.000' GROUP BY CustomerId
- Return the row set of the Customers (CustomerId) who made orders in year 2017 and total count of orders each Customer made.

With Having in Group By

- **Using Having Clause –**
  Having clause is used to filter values in Group By clause.
- The below query filters out some of the rows
- SELECT SalesOrderID, SUM(UnitPrice* OrderQty) AS TotalPrice FROM Sales.SalesOrderDetail GROUP BY SalesOrderID HAVING TotalPrice > 5000

Where and Having

- Since the WHERE clause’s visibility is one row at a time, there isn’t a way for it to evaluate the SUM across all SalesOrderID’s.
- The HAVING clause is evaluated after the grouping is created.
- Can use ‘Where’ clause with ‘Having’ clause as well.
- The WHERE clause is applied first to the individual rows in the tables.
- Only the rows that meet the conditions in the WHERE clause are grouped. The HAVING clause is then applied to the rows in the result set.

Having with Where

- SELECT SalesOrderID, SUM(UnitPrice * OrderQty) AS TotalPrice FROM Sales.SalesOrderDetail WHERE SalesOrderID > 500 GROUP BY SalesOrderID HAVING SUM(UnitPrice * OrderQty) > 10000
- The having clause will be applied on the rows that are filtered by where clause. Having clause can only compare results of aggregated functions or column part of the group by.

Having Where and GroupBy

- **WHERE** is used to filter records before any groupings take place that is *on single rows*.
- **GROUP BY** aggregates/ groups the rows and returns the summary *for each group*.
- **HAVING** is used to filter values after they *have been groups*.

PIVOT Usage in output

- **PIVOT** rotates a table-valued expression by turning the unique values from one column in the expression into multiple columns in the output.
- The **PIVOT** runs aggregations where they're required on any remaining column values that are wanted in the final output.

Pivot and Unpivot

- The PIVOT statement is used to convert table rows into columns.
- The UNPIVOT operator converts columns back to rows.
- **UNPIVOT** is another relational operator in **SQL** Server that performs almost the reverse operation of PIVOT, by rotating column values into rows values.

Reversing a PIVOT

- Reversing a PIVOT statement refers to the process of applying the UNPIVOT operator to the already PIVOTED dataset in order to retrieve the original dataset.
- UNPIVOT is a relational operator that accepts two columns (from a table or sub-query), along with a list of columns, and generates a row for each column specified in the list.
- In a query, it is specified in the FROM clause after the table name or sub-query.

Pivot Query with MS SQL Server

- A **pivot query** helps to summarize and re-visualize data in a table.
- For example a **pivot** table can help you see how many data points of a particular kind are present, or it can represent your data by aggregating it into different categories.
- The PIVOT and UNPIVOT relational operators to change a table-valued expression into another table.

Reversing a Pivot

- Reversing a PIVOT operator refers to the process of applying the UNPIVOT operator to a pivoted table in order to get back to the original table.
- Reversing Non-aggregate Pivoted Table.
- Reversing a PIVOT operator is only possible if the pivoted table doesn’t contain aggregated data.

Pivot with MySqL

- Rotate rows into columns in MySQL to transform data.
- MySQL does not have PIVOT function, so in order to rotate data from rows into columns you will have to use a CASE expression along with an aggregate function.
- MySQL does not have PIVOT function, so in order to rotate data from rows into columns, use a CASE expression along with an aggregate function.
- Use the aggregate function (SUM) with conditional logic
- select r.rep_name, sum(case when p.prod_name = 'Shoes' then s.quantity else **0** end) as Shoes.

Timeseries Database

- A **time-series database** (TSDB) is a computer system that is designed to store and retrieve data records that are part of a “**time series**,” which is a set of data points that are associated with timestamps.
- The timestamps provide a critical context for each of the data points in how they are related to others.
- Example : InfluxDB, Apache Prometheus, Apache TimescaleDB.

The time-series data

- The “time-series data” as a sequence of data points, measuring the same thing over time, stored in time order.
- A series of numeric values, each paired with a timestamp, defined by a name and a set of labeled dimensions (or “tags”).
- The sensors collecting data from three settings: a city, farm, and factory.
- The taxicab rides for the first few seconds of 2019.
- Each row is a “measurement” collected at a specific time.

Data over time

- Time series data are simply measurements or events that are tracked, monitored, downsampled, and aggregated over time.
- This could be server metrics, application performance monitoring, network data, sensor data, events, clicks, trades in a market, and many other types of analytics data.
- A time series database is built specifically for handling metrics and events or measurements that are time-stamped.

Time series query

- A time series query refers to one that finds, from a set of time series, the time series or subseries that satisfy a given search criteria.
- *Time series* are sequences of data points spaced at strictly increasing times.
- The search criteria are domain specific rules defined with time series statistics or models, temporal dependencies, similarity between time series or patterns, etc.

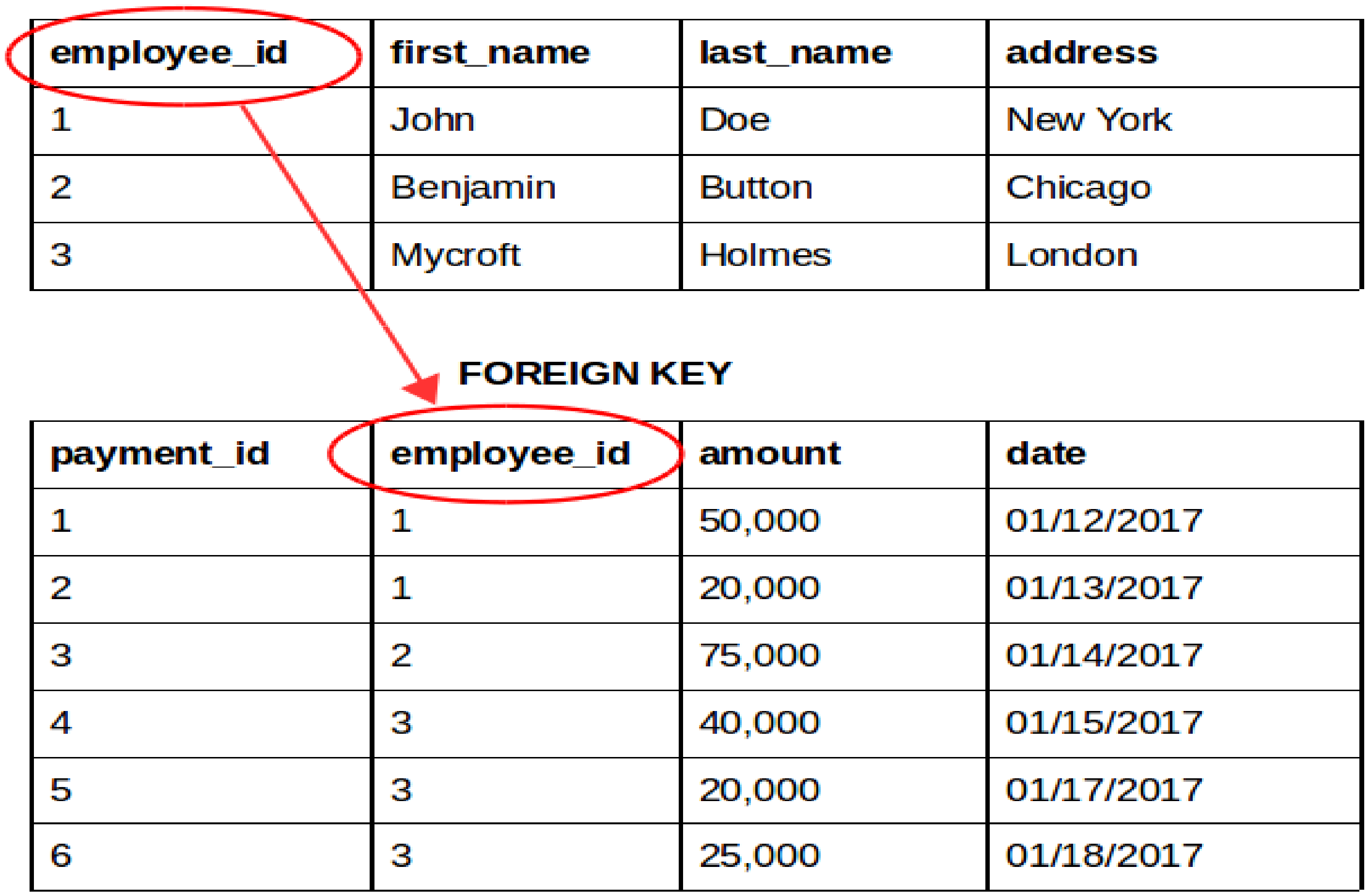
Time series data with Mysql

- MySQL and a number of it's variants can be used as a time-series database.
- MySQL provides a large range of functions for manipulating and pulling apart dates and time.
- Compared to other databases they are inconsistent and slow.

DB Schema for storage pattern

- RDBMS schema consists of several interrelated tables.
- The data is ordered in a structure with arranged *rows* and *columns* known as a *table* (relation).
- The rows are also referred as *tuples* whereas the columns are referred as *attributes*.
- The connection between these tables is known as a *relationship* and is represented by *foreign keys*.

Table Data Structure



employee_id	first_name	last_name	address
1	John	Doe	New York
2	Benjamin	Button	Chicago
3	Mycroft	Holmes	London

payment_id	employee_id	amount	date
1	1	50,000	01/12/2017
2	1	20,000	01/13/2017
3	2	75,000	01/14/2017
4	3	40,000	01/15/2017
5	3	20,000	01/17/2017
6	3	25,000	01/18/2017

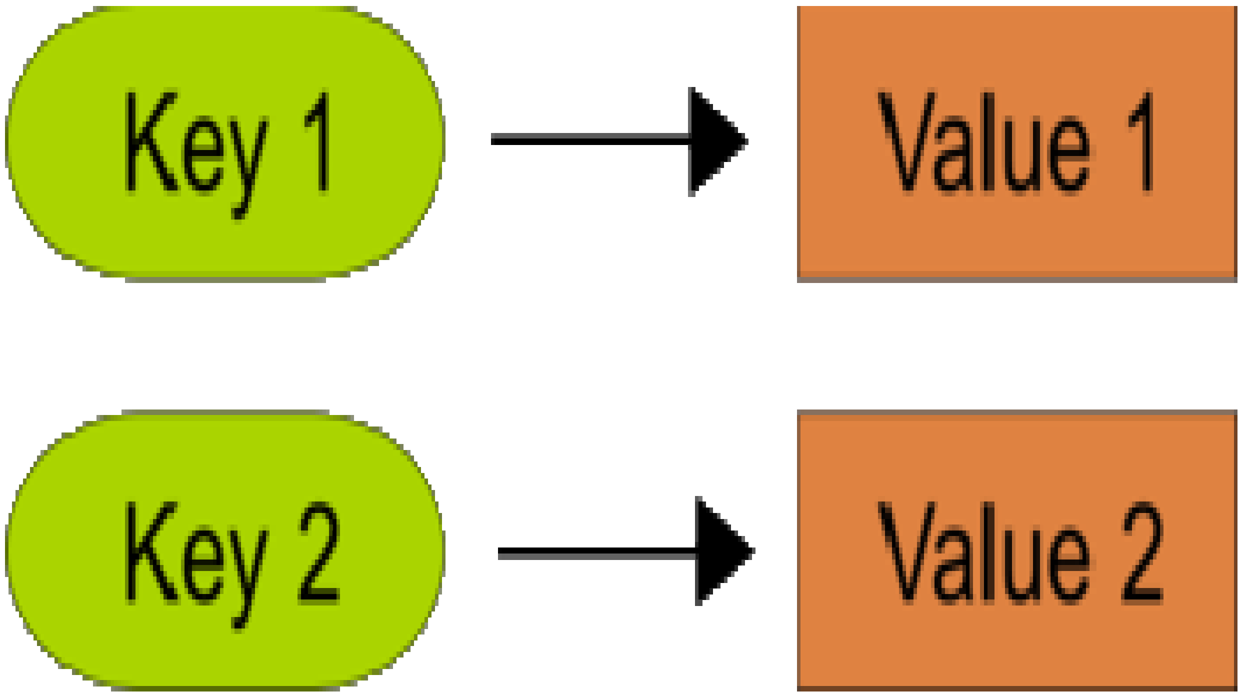
Store Configuration Data

- The single-row tables limits the number of configuration options you can have (since the number of columns in a row is usually limited).
- Every additional configuration option requires a DB schema change.

Key-Value Model

- In a name-value-pair table everything is "stringly typed" (you have to encode/decode your Boolean/Date/etc. parameters).
- The advantage of an entity-key-value model, is that you can readily add new keys -- and not have to affect the other entities that don't have those keys.

Key-Value structure



Key-Value Storage

- A key-value store is a database which uses an array of keys where each key is associated with only one value in a collection. It is quite similar to a dictionary or a map data structure.
- Key-value stores can be considered as the most primary and the simplest version of all the databases.
- It just has a one-way mapping from the key to the value to store data

Handle Key-Vale data

- The key-value stores usually do not have query languages as in RDBMS to retrieve data.
- They only provide some simple operations such as get, put and delete.
- Hence, the data querying (retrieving) should be handled manually at the application level.

Key-Value over RDBMS

- For application to be more performance driven, use key-value stores over RDBMS, since the key-value stores are capable of providing much higher performances than RDBMS.
- if the data model is not hierarchical you should consider using key-value stores over RDBMS.
- As the key-value stores are really simple and easy to handle, data can be modeled in a less complicated manner than in RDBMS.
- And it will end up having better performances than RDBMS as well.

Key-value Database

- A key-value database, or key-value store, is a data storage paradigm designed for storing, retrieving, and managing associative arrays.
- The data structure more commonly used as a dictionary or hash table.
- Dictionaries contain a collection of objects, or records, which in turn have many different fields within them, each containing data.

Querying with Key-Value Data

- The records are stored and retrieved using a key that uniquely identifies the record
- The key is used to find the data within the database.
- The key-value systems treat the data as a single opaque collection, which may have different fields for every record.

Key-Value Database

- NoSQL Data servers
- NoSQLz
- Coherence
- Couchbase
- Redis
- LevelDB
- Dynamo
- Oracle NoSQL

Track the Changes in Database

- The applications have access to insert, update, and delete out of that table.
- But due to the sensitive nature of that data, to have a quick and easy way to track who is doing what on that table.
- **Audit Tables** are used to track transactions against a particular table or tables.
  - Audit Tables are typically filled through the use of Database Triggers.
  - In other words, when X action happens on SensitiveInformation, insert the details of it in SensitiveInformationAudit.”

Audit the changes

- Manual audit
- SQL Server extended events
- SQL Server Triggers
- SQL Server data logs
- SQL Server profiling and traces
- Change tracking
- Temporal Tables

Temporal Table

- A system-versioned **temporal table** is a type of user **table** designed to keep a full history of data changes and allow easy point in time analysis.
- In system-versioned temporal table the period of validity for each row is managed by the system (i.e. database engine).

Temporal Structure

- Every temporal table has two explicitly defined columns, each with a **datetime2** data type.
- These columns are referred to as period columns. These period columns are used exclusively by the system to record period of validity for each row whenever a row is modified.

Keeping Track..

- The temporal table also contains a reference to another table with a mirrored schema.
- The system uses this table to automatically store the previous version of the row each time a row in the temporal table gets updated or deleted.
- This additional table is referred to as the history table, while the main table that stores current (actual) row versions is referred to as the current table or simply as the temporal table.

Data Changes Audit with Temporal Table

- Using normal table, application can retrieve current data.
- Using a system-versioned temporal table, you can retrieve data which was deleted or updated in the past.
- A temporal table will create a history table. The history table will store old data with "**start_time**" and "**end_time**".,Which indicates a time period for which the record was active.

Temporal Table Use case

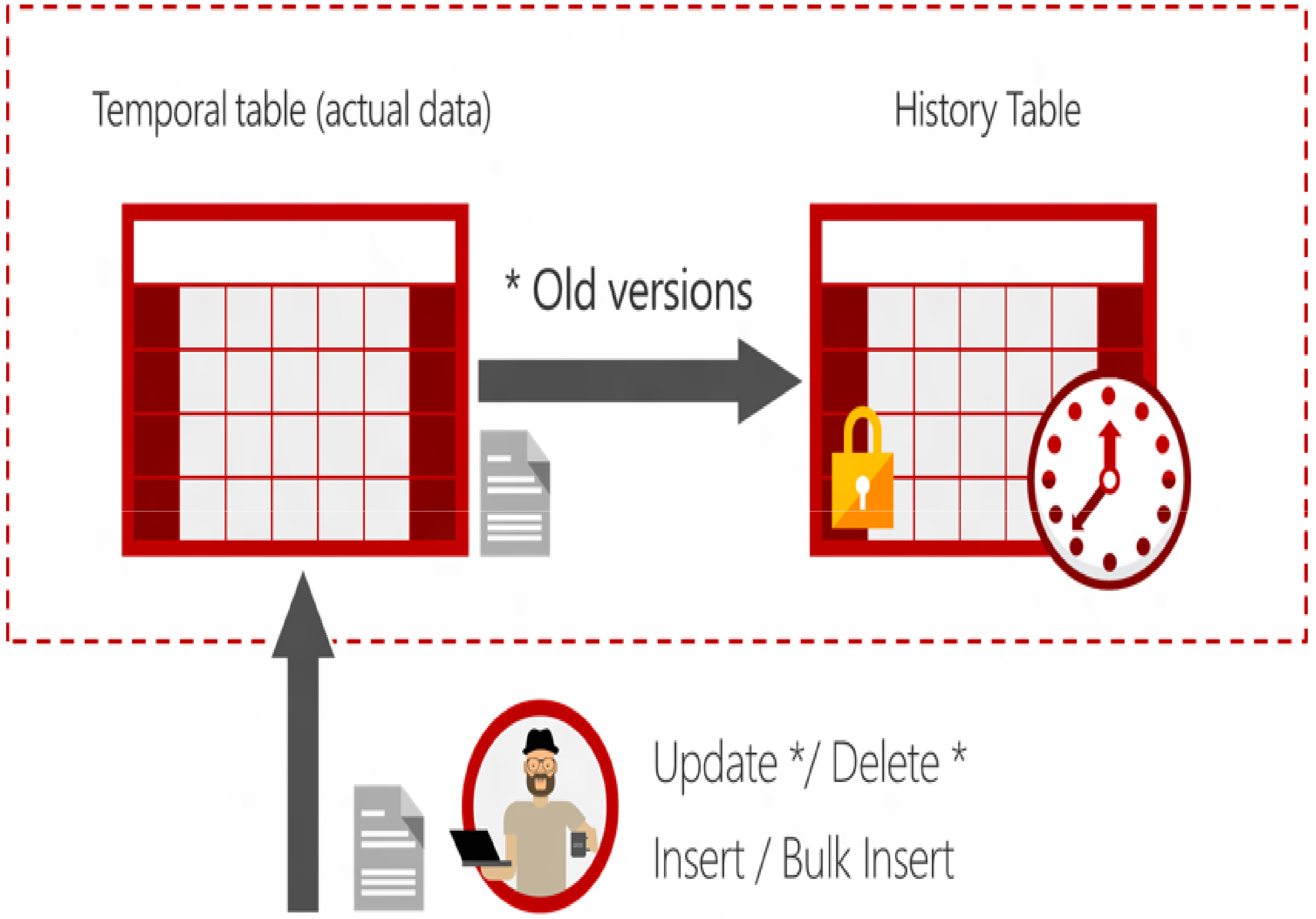
- Audit all data changes and perform data forensics when necessary
- Reconstruct state of the data as of any time in the past
- Calculate trends over time
- Maintain a slowly changing dimension for decision support applications
- Recover from accidental data changes and application errors

Temporal Structure

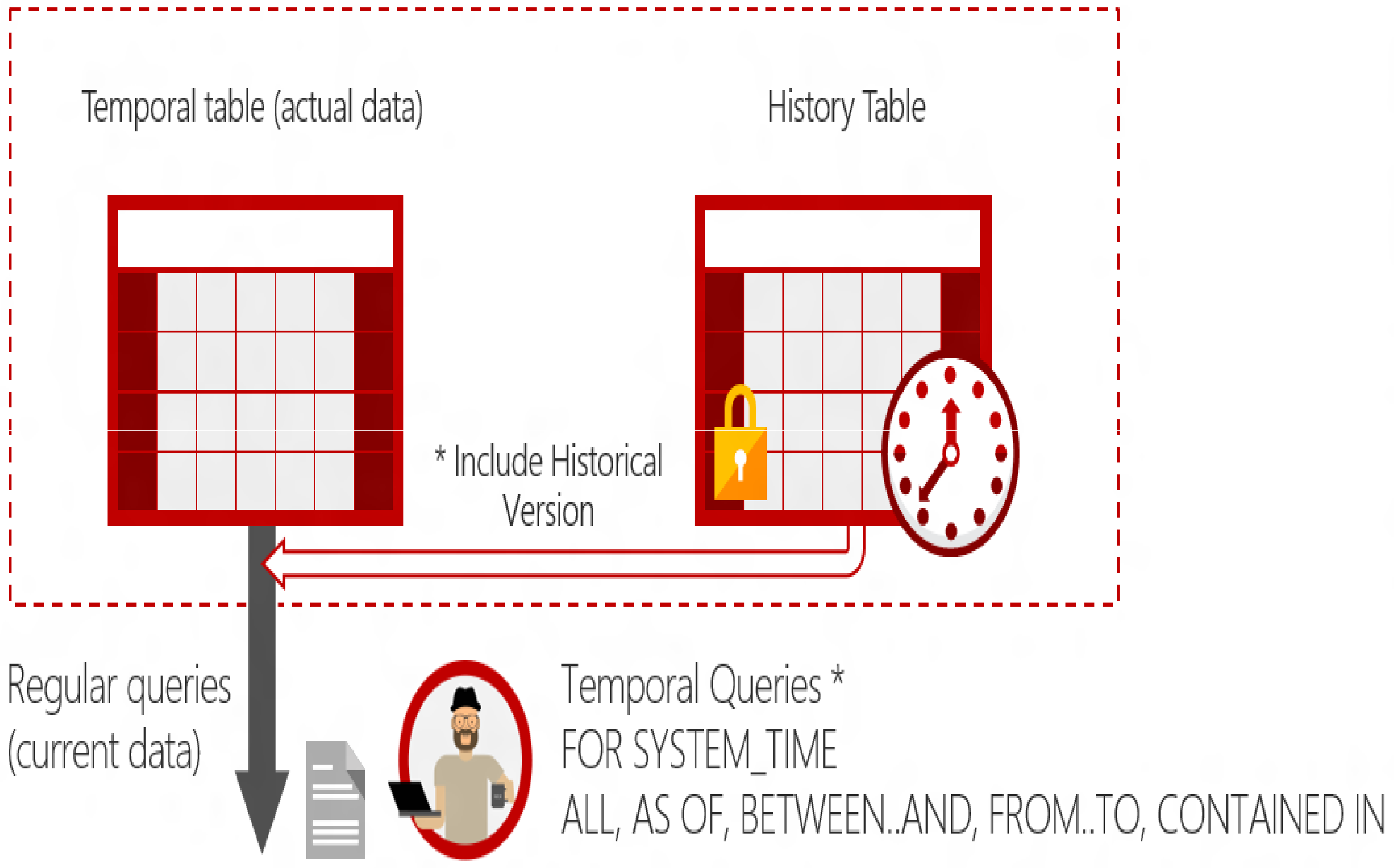
- System-versioning for a table is implemented as a pair of tables, a current table and a history table.
- Within each of these tables, the two additional **datetime2** columns are used to define the period of validity for each row.
  - Period start column: The system records the start time for the row in this column, typically denoted as the **SysStartTime** column.
  - Period end column: The system records the end time for the row in this column, typically denoted as the **SysEndTime** column.

Track of data and changes

- The current table contains the current value for each row.
- The history table contains each previous value for each row, if any, and the start time and end time for the period for which it was valid.


Temporal table (actual data)
History Table
* Old versions
Update */ Delete *
Insert / Bulk Insert

Query the temporal data


Temporal table (actual data)
History Table
* Include Historical Version
Regular queries (current data)
Temporal Queries *
FOR SYSTEM_TIME
ALL, AS OF, BETWEEN..AND, FROM..TO, CONTAINED IN

Query Plan

- A query on a huge table when performed without reading all the rows; a join involving several tables can be performed without comparing every combination of rows.
- The set of operations that the optimizer chooses to perform the most efficient query is called the “query execution plan”, also known as the EXPLAIN plan in MySQL.

Execution PLan

- Getting an Execution Plan
- Put explain in front of an SQL statement to retrieve the execution plan.
- **EXPLAIN** SELECT 1
- The EXPLAIN command provides information about how MySQL executes queries.
- EXPLAIN works with SELECT, DELETE, INSERT, REPLACE, and UPDATE statements.
  - Optimize the plan for quick execution

Index for quick fetch

- A database index is a data structure that improves the speed of operations in a **table**.
- Indexes can be created using one or more columns, providing the basis for both rapid random lookups and efficient ordering of access to records.
- Indexes are used to find rows with specific column values quickly.

Without Index

- Without an index, for query execution, the MySQL must begin with the first row and then read through the entire table to find the relevant rows.
- The larger the table, the more this costs.
- If the table has an index for the columns in question, MySQL can quickly determine the position to seek to in the middle of the data file without having to look at all the data.
- This is much faster than reading every row sequentially.

Index Creation

- While creating index, it should be taken into consideration which all columns will be used to make SQL queries and create one or more indexes on those columns.
- Practically, indexes are also a type of tables, which keep primary key or index field and a pointer to each record into the actual table.
- The users cannot see the indexes, they are just used to speed up queries and will be used by the Database Search Engine to locate records very fast.

Database partition

Partitioning is a way in which a database (MySQL in this case) splits its actual data down into separate tables, but still get treated as a single table by the SQL layer.

Partition Usage

- When partitioning in MySQL, it’s a good idea to find a natural partition key, to ensure that table lookups go to the correct partition or group of partitions.
- This means that all SELECT, UPDATE, DELETE should include that column in the WHERE clause.
- Otherwise, the storage engine does a scatter-gather, and queries ALL partitions in a UNION that is not concurrent

Partition key

- Generally, you must add the partition key into the primary key along with the auto increment, i.e., PRIMARY KEY (part_id,id).
- If you don’t have well-designed and small columns for this composite primary key, it could enlarge all of your secondary indexes.

Partitioning advanatge

- Partitioning makes it possible to store more data in one table than can be held on a single disk or file system partition.
- Data that loses its usefulness can often be easily removed from a partitioned table by dropping the partition (or partitions) containing only that data.
- Conversely, the process of adding new data can in some cases be done by adding one or more new partitions for storing specifically that data.

Search with partition

- Some queries can be optimized in virtue of the fact that data satisfying a given WHERE clause can be stored only on one or more partitions, which automatically excludes any remaining partitions from the search.
- MySQL supports explicit partition selection for queries.
  - SELECT * FROM t PARTITION (p0,p1) WHERE c < 5 selects only those rows in partitions p0 and p1 that match the WHERE condition

Horizontal vs. Vertical Partitioning

- Horizontal partitioning means that all rows matching the partitioning function will be assigned to different physical partitions.
- Vertical partitioning allows different table columns to be split into different physical partitions.
  - *Vertical partitioning is about splitting up columns*
  - *Horizontal partitioning is about splitting up rows*
- Currently, MySQL supports horizontal partitioning but not vertical.

Partition Types

- RANGE Partitioning: assigns rows to partitions based on column values that fall within a stated range. The values should be contiguous, but they should not overlap each other.
- LIST Partitioning: similar to RANGE, except that the partition is selected based on columns matching one of a set of discrete values.

HASH Partitioning

- A partition is selected based on the value returned by a user-defined expression.
- This expression operates on column values in rows that will be inserted into the table.
  A HASH partition expression can consist of any valid MySQL expression that yields a nonnegative integer value.
- HASH is used mainly to evenly distribute data among the number of partitions the user has chosen.

Storage with Hash

- For RANGE and LIST, one must define the partitions where the data will be stored.
- HASH does this automatically, based on the expression or INT value of the selected column.

KEY Partitioning

- This is very similar to HASH partitioning, but the hashing function is supplied by MySQL.
- A KEY partition can specify zero or many columns, which can contain non-integer values.
- An integer result will be returned regardless of the column data type.
- MySQL will automatically use the primary key or a unique key as the partitioning column. If no unique keys are available, the statement will fail.

Partition Pruning

- Don’t search partitions where there is nothing to be found.
- For select queries the optimizer kicks in and leave unnecessary partitions out of the search and hence the query is more efficient.
- The optimizer automatically “prunes” only when the WHERE condition can be simplified
  - partition_column = constant
  - partition_column IN (constant1, constant2, ..., constant-N)
  - also explicitly select the partition like
  - SELECT * FROM samplelogs PARTITION (currentlogs) WHERE created > '2016-01-01';

Partition Names

- When partition names are generated by MySQL (as in HASH and KEY partitions) the name pattern is p0, p1,..., p*N-1* (where *N* is the number of partitions).
- Partition selection is supported for most operations like DELETE, UPDATE, INSERT, JOIN, etc.

Database optimization

- **Database optimization** involves maximizing the speed and efficiency with which data is retrieved.
- **Database** designers, administrators and analysts work together to **optimize** system performance through diverse methods.

Optimization Techniques

- **Proper indexing: I**ndex is a data structure that helps speed up the data retrieval process overall.
- Unique index is a kind of indexing that creates separate data columns without overlapping each other.
- Without any indexing at all, the processing is very slow, whereas indexing everything will render the insert and update triggers ineffective.
- Proper indexing ensures quicker access to the database.
- Excessive indexing or no indexing at all are both wrong.

Techniques..

- **Retrieve the relevant data only**
- Specifying the data that one requires enables precision in retrieval.
- Using the commands * and LIMIT, instead of SELECT * as and when required is a great way of tuning the database, while avoiding retrieving the whole set of data when the user wants only a certain part of it.
- Of course, it is not be necessary when the amount of data overall is less.
- But when accessing data from a large source, specifying the portions required would save a lot of essential time.

Techniques..

- **Getting rid of correlated sub-queries**
- A correlated sub-query depends on the parent or outer query.
- This kind of search is done row by row and it decreases the overall speed of the process.
- This problem usually lies in the command of WHERE from the outer query, applying which, the sub-query runs for each row, returned by the parent query, consequently slowing the whole process and reducing the efficiency of the database.
- So, a better way of tuning the database, in this case, is to the INNER JOIN command, instead of the correlated sub-query

Techniques

- **Avoid temporary tables: I**f any code can be well written in a simple way, there is absolutely no need to make it complex with temporary tables.
- Of course, if a data has a specific procedure to be set up which requires multiple queries, the use of temporary tables in such cases are, in fact, recommended.
- Temporary tables are often alternated by sub-queries, but one has to keep in mind the specific efficiency that each of these would provide in separate cases.

Techniques..

- **Avoid coding loops:** is very much needed in order to avoid slowing down of the whole sequence.
- This is achieved by using the unique UPDATE or INSERT commands with individual rows, and by ensuring that the command WHERE does not update the stored data in case it finds a matching preexisting data.

Techniques..

- **Execution plans: T**he execution plan tool created by the optimizer play major role in tuning SQL databases.
- They help in creating proper indexes too.
- Although, its main function is to display graphically the various methods to retrieve data.
- This, in turn, helps in creating the needed indexes and doing the other required steps to optimize the database.

Techniques..

- Optimizing Data Size: Design the tables to minimize their space on the disk.
- Optimize size with
  - Table Columns
  - Row Format
  - Indexes
  - Joins
  - Normalization

Optimize

- Optimize the MYSQL Data Types
  - Numeric Data
  - Character and String Type
  - BLOB Types
- Optimize the sql statements with queries and sub-queries
- Optimize indexes
- Optimize on Locking and Caching