



# DATABASE

Internals

# PROBLEM : MOVIE SITE

IMDB wants to revamp User Experience on their site by enhancing database performance and supporting some new features. They have approached our team to build a database solution to answer following queries quickly.

- Movie information such as title, director, cast, release data, rating.
- Fetch list of movies filtered by director.
- Fetch list of movies filtered by genre and release year.
- Fetch other movies and role, played by cast of specified movie.
- Similar movies.
- Highest rated movie by genre for specified year.
- Average rating of movies in specified year.
- Suggestion while typing movie, cast and director names.

They are willing to let us use around 10 computers with following specification. (256GB Ram, 2TB SSD, 8 Cores @3.14GHz Processor).

They are currently using a MySQL database and have provided us with schema of tables and number of entries in each table. They expect our database to support addition of new data. However, primary concern is returning query result quickly as read/write ratio is approx. 100000:1.

# PROBLEM : MOVIE SITE

IMDB hosts an event where an actor/actress participate. During this event, they ask a question and viewers of this event can answer this question correctly to win a prize. Every 1000th user with correct answer is considered a winner, until 100th winner is declared.

They have told our team that it'd be great if our database solution provide capabilities to support storage of user's answer along-with User details if they're a winner or not.

# EXPECTATIONS FROM DATABASE

What is the need of a database ?

Why isn't data simply written to file ?

# EXPECTATIONS FROM DATABASE

## Scalability

- Upgrading existing system
- Adding more machines

## Schema evolution

## Consistency

- Latest write
- Quorum requirement

## Fault tolerant

- Write Ahead Logs
- Snapshot
- Copy-on-write

- Highly available
  - Replication
- Performant
  - Concurrent
  - Fast lookup
  - Quick write
- Durable
  - Write to HDD or SSD
- Security
  - Document / Table level access control
  - Access on View or Materialized view
  - Row-level access control

# COMPONENTS OF DATABASE

- Storage
- Data Model (Schema)
- Database Tuning (Algorithm & Data Structure)
- Query Engine

# DATA STORAGE

Where should data be written ?

How should data be written ?

- Should it be written in a continuous location ?
- Should it be written in human friendly or computer friendly manner ?

How quickly can data be read ?

How easily can data be updated ?

How difficult would adding new columns or removing columns or changing datatype be ?

# DATA STORAGE

## Physical Store

- RAM
- SSD
- Hard Disk

## Locality

- Document
- Row Oriented
- Column Oriented
- Linked List (Graph)

## Ordering

- Ordered
  - LSM
  - B/B+ Tree
- Unordered
  - Append

## Data encoding & decoding

- Human Readable
  - JSON
  - CSV
  - XML
- Binary Objects
  - Thrift
  - Avro
  - Parquet



# PHYSICAL STORE

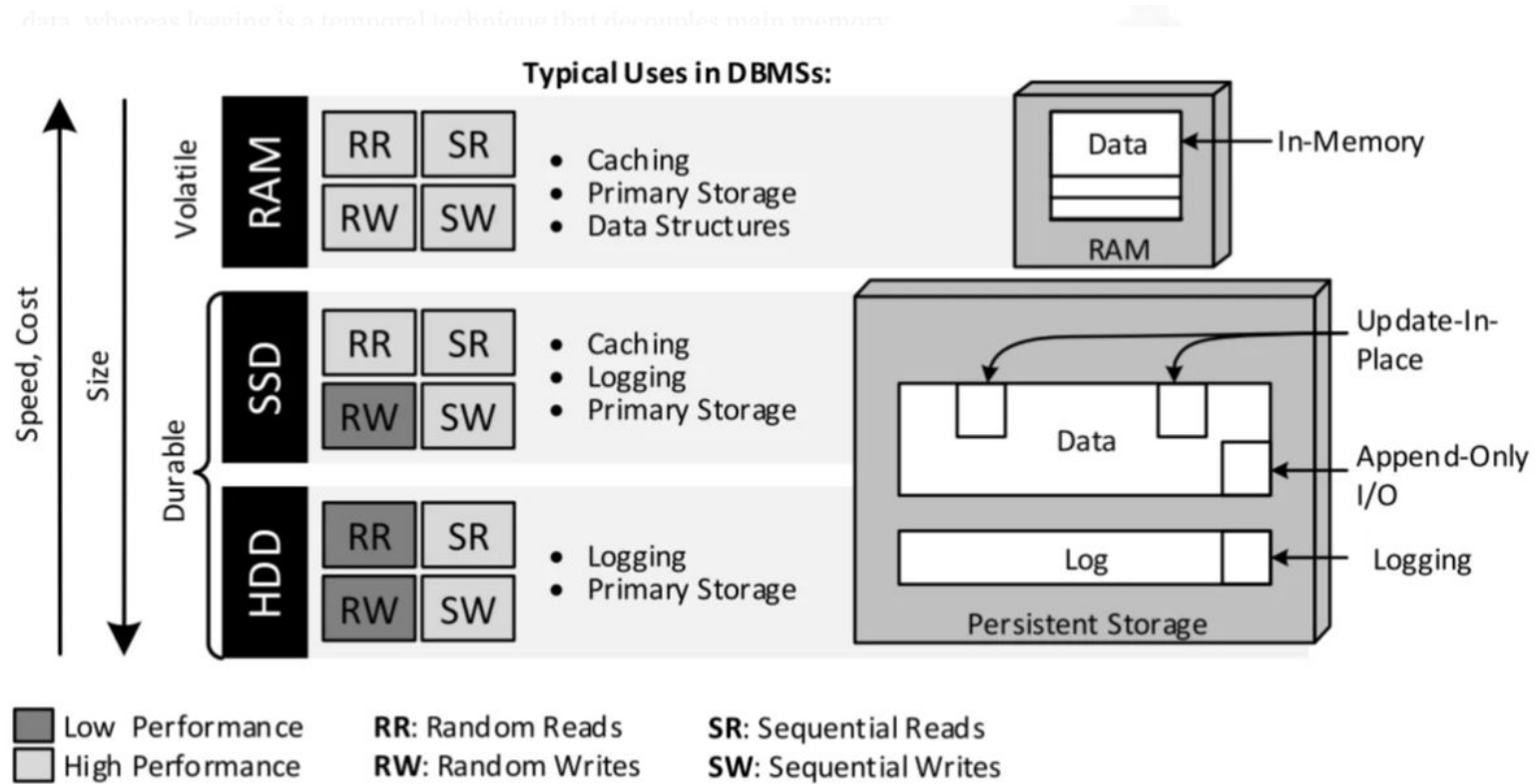
The physical store plays a crucial role in database performance, durability and capacity.

Different storage system has different method of data access.

Data Structures and Algorithms suitable for reading or writing data in various system differs.

	Sequential Read (MB/s)	Sequential Write (MB/s)	Random Read (MB/s)	Random Write (MB/s)	Persistence
RAM	21400	16600	3200	3100	No
SSD	3400	2900	1200	1200	Yes
HDD	58	38	1	0.9	Yes

# PHYSICAL STORE



# LOCALITY

Data locality determines the ease of accessing related data.

The physical store and its storage management, are optimized for certain access layout.

Physical data locality can be achieved by storing data closer on physical store.

Virtual data locality can be achieved by storing references of related data together.

Different methods of keeping related data together:

- Document
- Row Oriented
- Column Oriented
- Linked List (Graph)
- Key - Value

# LOCALITY : DOCUMENT

Data is treated as document with different structures. Enables easy retrieval and provides high availability.  
Best suited for storing online profiles or use-cases requiring evolving data-models.

```
{
  "ID" : 123,
  "Name" : "John Doe",
  "EmploymentData": [
    {
      "EmployerName" : "XYZCompany",
      "StartDate" : 2000-01-01,
      "EndDate" : 2000-07-01,
      "CTC" : 2300000,
      "AssociatedWork": [
        "Department" : "D365",
        "WorkDetail" : "Optimizing business process for enterprise customers"
      ]
    },...
  ]
}
```

# LOCALITY : ROW ORIENTED

Data for a row is stored together. Addition of a new record is a simple append operation.

Best suited for structured data, data models with relationships or transactions.

EmployeeTable

[{"ID": "123", "Name": "John Doe"}, {"ID": "124", "Name": "Nick Cooper"}, ...]

EmploymentDataTable

[{"EmploymentDataID": 1, "EmployerName": "XYZCompany", ...}, ...]

AssociatedWorkTable

[{"AssociatedWorkID": 1, "Department": "D365", "WorkDetail": "Optimizing business process for enterprise customers"}]

EmployeeEmploymentTable

[{"EmployeeID": 123, "EmploymentDataID": 1}, ...]

EmploymentDataAssociatedWorkTable

[{"EmploymentDataID": 1, "AssociatedWorkID": 1}, ...]

# LOCALITY : COLUMN ORIENTED

Data of a column is stored together. Reduces I/O by selectively searching columns.

Number of unique values in particular column is often much less than total number of records. This provides an opportunity to compress data.

It is also suitable for vectorization, and thus enhances performance.

Best suited for Analytical queries, use-cases where only subset of columns are required to answer query or data warehouse.

EmployeeIDCollection

[123, 176, 190, 1200;...]

EmployeeNameCollection

["John Doe","Alex Norman","Sofia Nuggets",,..]

# LOCALITY : GRAPH

Data is stored as graphs, which helps efficiently represent and analyze interrelations in data. Nodes are entities and relationship is defined by edges. This entire structure is like LinkedList. Best suited for Social Networks, Supply Chain, Knowledge graph, Recommendation engine.

<Properties>

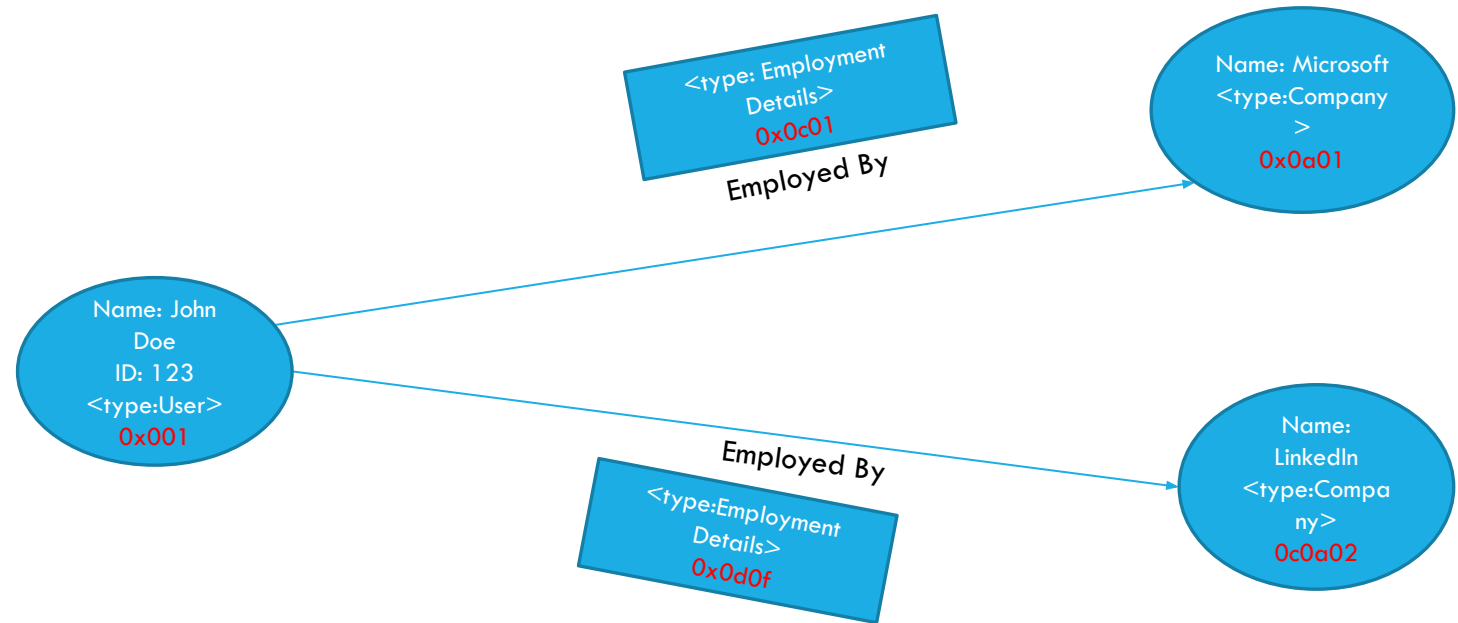
[prevPtr; key; value; nextPtr]

<Nodes>

[prevPtr; propertyPtr; relationshipPtr; nextPtr]

<Relationships>

[startNode; propertyPtr; endNode; startNodePrevRelationshipPtr, startNodeNextRelationshipPtr, endNodePrevRelationshipPtr, endNodeNextRelationshipPtr]



# DATA ORDERING

There are several records in database.

These records can be in random order or with some ordering.

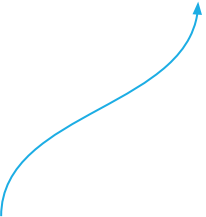
While simply writing record at any available space is convenient during write operation, ordered data makes it easier to locate record among all records.

Data ordering can be achieved by physically ordering records as per some order or by creating a virtual mechanism to be able to access unordered data in ordered way.



# DATA ORDERING : APPEND

RowID	Name	Employer
1	John Doe	Microsoft
2	Sofia Nuggets	LinkedIn

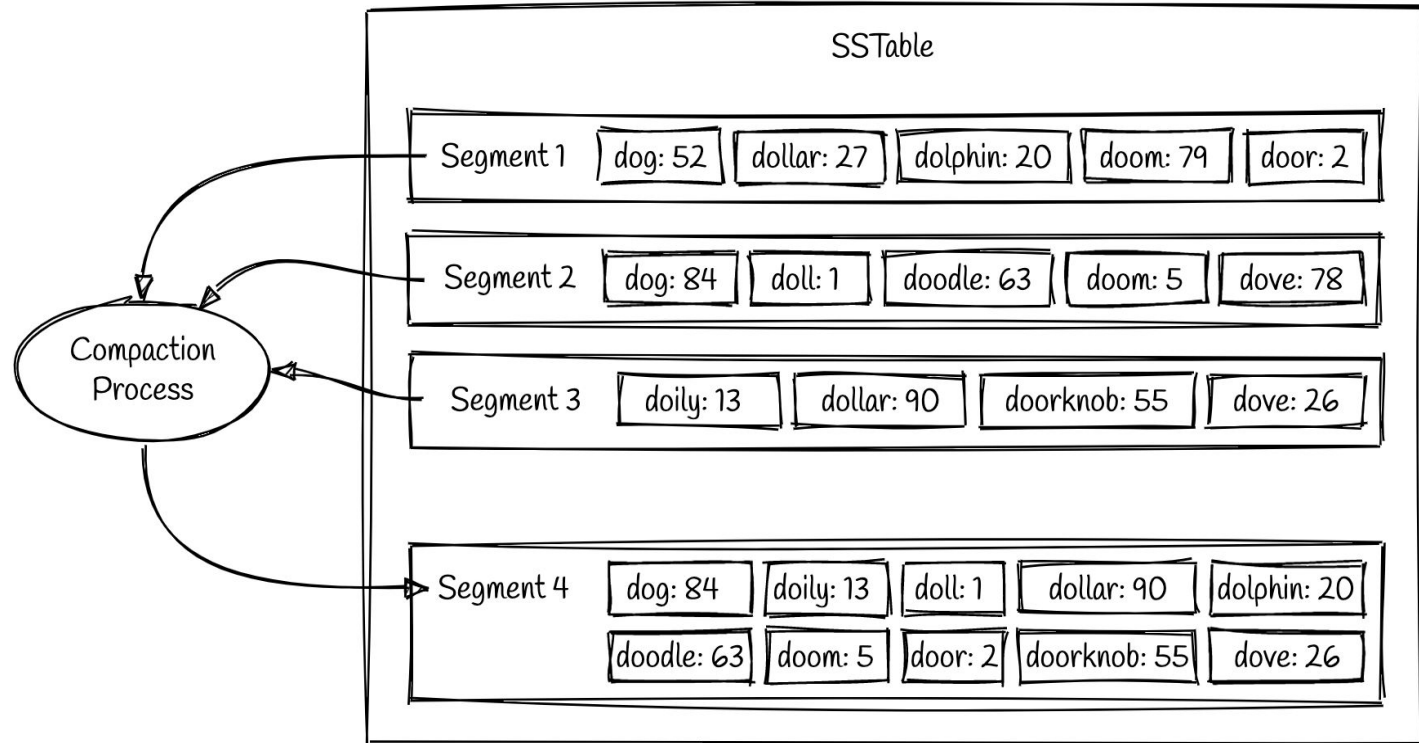


RowID	Name	Employer
3	Alex Raymond	Github

# DATA ORDERING: LSM TREE

Data is in sorted order.

A hash-table with limited keys can be used to quickly locate file containing required key.



# DATA ENCODING & DECODING

Data encoding determines the size of data when stored on disk or transferred over network.  
The ease of making changes to schema varies significantly with different data encoding format.

- Human Readable

  - JSON

  - CSV

  - XML

- Binary Objects

  - Thrift

  - Avro

  - Parquet

# DATA ENCODING : JSON

It is human readable and hence easy to debug.

New fields can be added or removed easily.

Optional support for schema.

REST generally uses JSON.

```
{
  "userName": "Martin",
  "favoriteNumber": 1337,
  "interests": ["daydreaming", "hacking"]
}
```

Bytes : 81

MessagePack

Byte sequence (66 bytes):

83	a8	75	73	65	72	4e	61	6d	65	a6	4d	61	72	74	69	6e	ae	66	61
76	6f	72	69	74	65	4e	75	6d	62	65	72	cd	05	39	a9	69	6e	74	65
72	65	73	74	73	92	ab	64	61	79	64	72	65	61	6d	69	6e	67	a7	68
61	63	6b	69	6e	67														

Breakdown:

object (3 entries)	string (length 8)	u s e r N a m e	string (length 6)	M a r t i n
83	a8	75 73 65 72 4e 61 6d 65	a6	4d 61 72 74 69 6e
	string (length 14)	f a v o r i t e N u m b e r		
	ae	66 61 76 6f 72 69 74 65 4e 75 6d 62 65 72		
	uint16	1337	string (length 9)	i n t e r e s t s
	cd	05 39	a9	69 6e 74 65 72 65 73 74 73
array (2 entries)	string (length 11)	d a y d r e a m i n g		
92	ab	64 61 79 64 72 65 61 6d 69 6e 67		
	string (length 7)	h a c k i n g		
	a7	68 61 63 6b 69 6e 67		

# DATA ENCODING : XML

It is human readable and hence easy to debug. It is too verbose.

New fields can be added or removed easily.

Optional support for schema.

SOAP uses XML for request and response.

```
<?xml version="1.0" encoding="UTF-8" ?>
<root>
  <userName>Martin</userName>
  <favoriteNumber>1337</favoriteNumber>
  <interests>daydreaming</interests>
  <interests>hacking</interests>
</root>
```

Bytes : 155 (excluding header)

# DATA ENCODING : CSV

It is human readable and hence easy to debug.

Addition of new fields requires updating previous records or creation of new file with updated header.

Schema can be enforced by application after data has been read.

CSV generally has issues with delimiter as different implementations work differently.

```
username, favoriteNumber, Interests
```

```
Martin, 1337, daydreaming | hacking
```

Bytes : 31 (excluding header)

# DATA ENCODING : AVRO

Avro uses a schema to specify the structure of the data being encoded.

In the encoded data, there is no reference to fields or tags (i.e. order of field in schema).

Avro uses writer schema and reader schema to encode and decode data.

Reader schema is matched against writer schema to determine values corresponding to field.

Writer's schema for Person record

Datatype	Field name
string	userName
union {null, long}	favoriteNumber
array<string>	interests
string	photoURL

Reader's schema for Person record

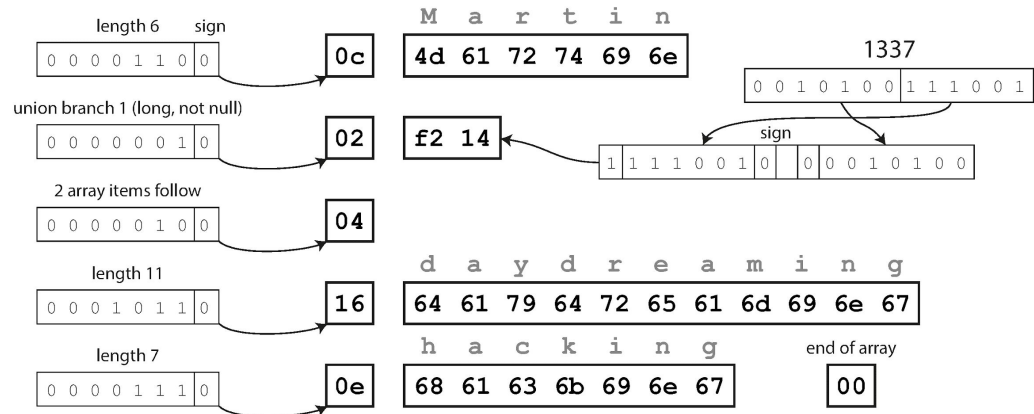
Datatype	Field name
long	userID
union {null, int}	favoriteNumber
string	userName
array<string>	interests

Avro

Byte sequence (32 bytes):

0c	4d 61 72 74 69 6e	02	f2 14	04	16	64 61 79 64 72 65 61 6d
69 6e 67	0e	68 61 63 6b 69 6e 67	00			

Breakdown:



# ALGORITHM & DATA STRUCTURE

How can databases handle large amount of data ?

Is there different ways to store old (rarely used) data and latest (frequently queried) data ?

How many copies of data exists ?

Does a single system store all data, what happens in case of system failure ?

Are data files only stored on disk ?

How to quickly get to data relevant to query among all data stored in database ?



# ALGORITHM & DATA STRUCTURE

Compression

Sharding

- Vertical Partitioning
- Horizontal Partitioning

Replication

Buffer / Page Pool

Cache

Index (In-memory or Disk)

- Hash table
- Key-Value
  - B/B+ tree
  - LSM tree
- Clustered/Non-Clustered
- Multi-Column
- Quad-tree

# COMPRESSION

I/O is the main bottleneck if DBMS has to fetch data from Disk.

Compression techniques reduces the data size. Trade-off is between speed vs compression-ratio.

Several compression techniques can be combined together for greater compression.

- Prefix Compression
- Dictionary Compression
- Row Compression
- Bitmap Encoding
- Null Suppression
- Order Preserving Compression

# COMPRESSION : PREFIX

In each column, a column prefix is used to encode the values.

Page Header		
aaabb	aaaab	abcd
aaabcc	bbbb	abcd
aaaccc	aaaacc	bbbb

Page Header		
aaabcc	aaaacc	abcd
4b	4b	[empty]
[empty]	[0bbbb]	[empty]
3ccc	[empty]	[0bbbb]

# COMPRESSION : DICTIONARY

A symbol table or a dictionary is created and used as replacement of commonly appearing values. It can be applied on both column as well as row.

Page Header		
aaabb	aaaab	abcd
aaabcc	bbbb	abcd
aaaccc	aaaacc	bbbb

Page Header		
aaabcc	aaaacc	abcd
4b	[0bbbb]	
0	0	[empty]
[empty]	1	[empty]
3ccc	[empty]	1

# COMPRESSION : ROW

Data in a row can be closely packed.

E.g. Consider a value 10 is present in a row. Instead of storing it as 4-byte integer, only 1 byte can be used to store it as binary representation of 10 i.e. 1010 requires only 4 bits and additional 4 bits can be used to store metadata.

E.g. Consider a column Name defined as Varchar(50). A row containing the value “John Doe” needs only 9 Bytes to store it.

# COMPRESSION : BITMAP

If the unique set of values in a column is extremely low, bitmap encoding can be used.

E.g. Consider the column “States”, it can have only 29 different values. Instead, of storing the values such as “Telangana”, “Arunachal Pradesh”, a bitmap of size 29 can be used. An offset can be assigned to each state i.e. 1st bit corresponds to Arunachal Pradesh, 23rd bit corresponds to Telangana. Now, columns would contain a bit-vector of size 29 with bit corresponding to their values set.

City	State
Hyderabad	Telangana
Itanagar	Arunachal Pradesh
Bangalore	Karnataka

City	State
Hyderabad	[00000000000000000000000001000000]
Itanagar	[10000000000000000000000000000000]
Bangalore	...

# SHARDING

Sharding or partitioning refers to breaking down data into smaller chunks.

Data decomposition can be done in two ways:

- Horizontal
- Vertical

Original Table

CUSTOMER ID	FIRST NAME	LAST NAME	FAVORITE COLOR
1	TAEKO	OHNUKI	BLUE
2	O.V.	WRIGHT	GREEN
3	SELDA	BAĞCAN	PURPLE
4	JIM	PEPPER	AUBERGINE

Vertical Partitions

VP1

CUSTOMER ID	FIRST NAME	LAST NAME
1	TAEKO	OHNUKI
2	O.V.	WRIGHT
3	SELDA	BAĞCAN
4	JIM	PEPPER

VP2

CUSTOMER ID	FAVORITE COLOR
1	BLUE
2	GREEN
3	PURPLE
4	AUBERGINE

Horizontal Partitions

HP1

CUSTOMER ID	FIRST NAME	LAST NAME	FAVORITE COLOR
1	TAEKO	OHNUKI	BLUE
2	O.V.	WRIGHT	GREEN

HP2

CUSTOMER ID	FIRST NAME	LAST NAME	FAVORITE COLOR
3	SELDA	BAĞCAN	PURPLE
4	JIM	PEPPER	AUBERGINE

# SHARDING : HORIZONTAL

Horizontal sharding decomposes tables into multiple smaller table such that they all have same schema but only a unique subset of original records are present in each of them.

The value from a column or set of columns is used to create a shard key. Shard key can be used with a Hash function to obtain shard information. Hash function can be programmed to return i. Random value ii. Map a range into same shard iii. Static mapping.

## □ Key Based

A column or set of columns is hashed to determine a key. This key decided the shard on which data would be present.

## □ Range Based

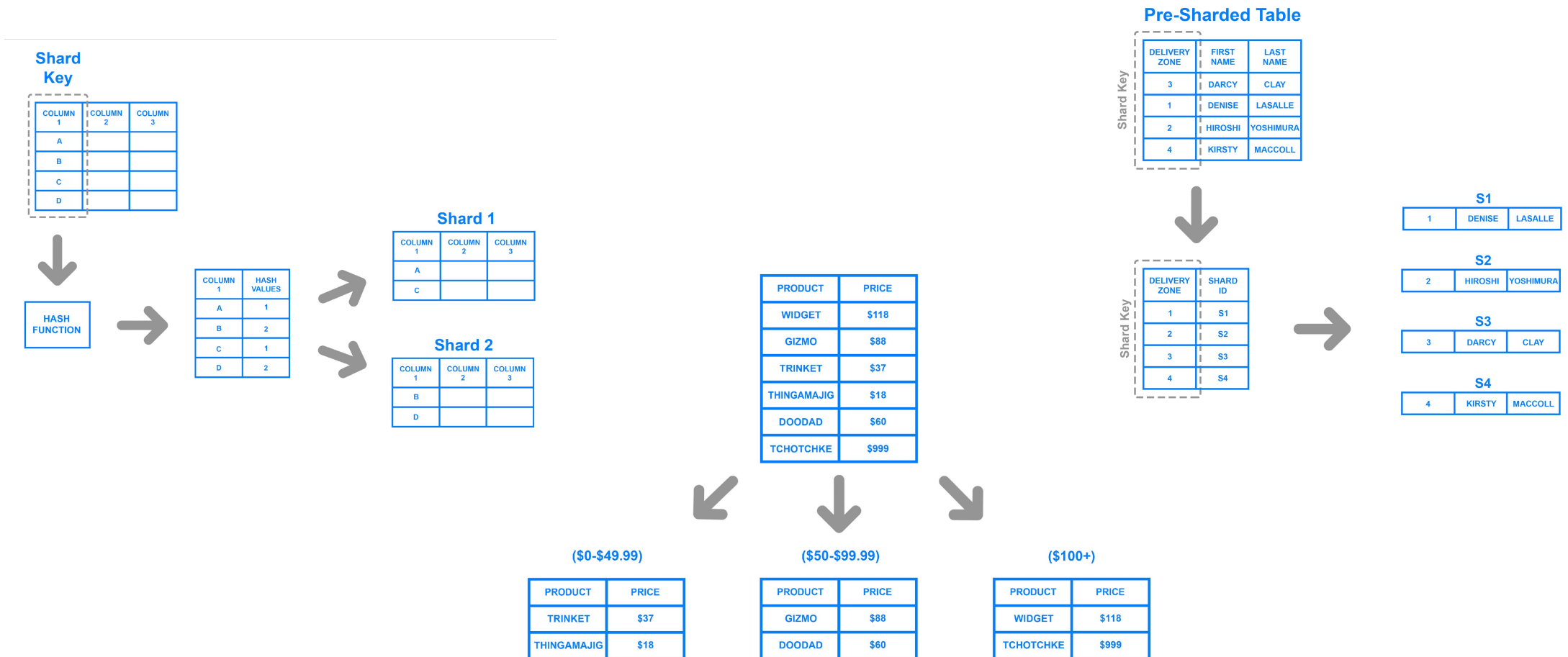
A range of values corresponding to a column is used to determine the shard.

## □ Directory Based

A lookup table is used to obtain the shard detail. Typically, values from a column is used as key to lookup table.



# SHARDING : HORIZONTAL



# SHARDING : VERTICAL

Vertical sharding decomposes table into different tables such that only a subset of columns are present in each table.

Normalization is one of ways to break down tables in above manner. However, even normalized tables can be split vertically so that only certain columns are present in one table.

ID	Name	Age	Address
1	Alex Russel	19	St. Louis
2	Grammy Mars	23	Atlanta

ID	Name	Age
1	Alex Russel	19
2	Grammy Mars	23

ID	Address
1	St. Louis
2	Atlanta

# REPLICATION

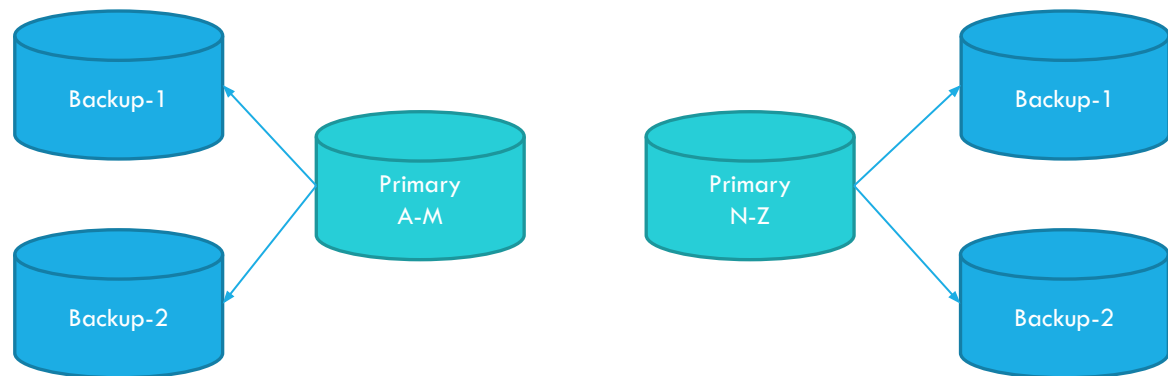
Replication in database adds redundancy to system.

Replicas are useful to switchover in case of failure of node. However, there is added complexity in system to ensure consistency among all replicas.

Replicas are also useful to improve query performance by load balancing. Replicas across geography also reduces network latency.

Various replication mechanisms

- Log Based
- Full Table
- Key-Based Incremental



# REPLICATION : LOG BASED

## Statement Based

- Queries or actions which modify the database are stored and re-executed on replicas in same order.
- Logs are smaller in size compared to row based, a system can be restored to state before failure by executing the statements present in log.
- In case queries contain Hardware or OS dependent value, a replica with different hardware or OS may produce different result.

## Row Based

- Database logs all updated rows. Replicas iterate over these records in log and perform update.
- Queries which affect lot of rows result in lots of records written to log files, logs file could become extremely large.
- It is one of most effective and safe replication method.

# REPLICATION : FULL TABLE

## Snapshot

- ❑ Full table data is snapshotted at intervals and distributed for replication
- ❑ Suitable for smaller tables e.g., Web Servers

## Transactional

- ❑ Updates on master database is monitored and synced with replicas
- ❑ Suitable for larger tables
- ❑ Since replicas stores all changes but with slight delay, this is more suitable for queries on historical data

# REPLICATION : KEY-BASED INCREMENTAL

The updates to primary database are chained and stores change based on unique Id or keys. Only the final updated value is synced with other databases.

It is suitable for use-cases having very frequent update on specific keys.

Replicas doesn't contain historical changes, hence this system is more suitable for latest-value queries.

# BUFFER POOL

Disk I/O is expensive.

Fetching pages from disk takes time.

Database Systems creates a pool of pages in memory. This space is used to store pages from disk and perform required operation.

Buffer pool can store hot pages to efficiently perform most queries.

Various algorithms are used to read-ahead (prefetch) pages from disk to buffer, evict pages from buffer and write back update to disk.

# INDEX

Index helps to quickly locate data of interest.

A query would have to scan all table data. With information from index, the search space can be reduced.

Index can also be considered as data for Index table and needs to be stored!

The choice of index, depends on data storage layout as well as kind of query.

Some indexing mechanism can quickly locate a piece of data, others can reduce the search space for range based queries.

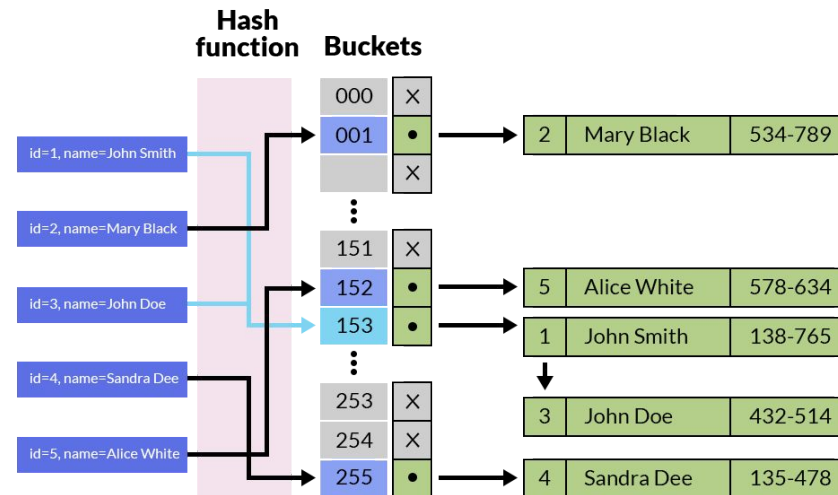


# INDEX : HASH TABLE

Hash based indexing uses a hash function to map “key” to an index in hash-table.

For each key, the value corresponds to page offset in disk.

Hash-index is typically located in memory. However, they can also be stored on disk with trade-off in performance.

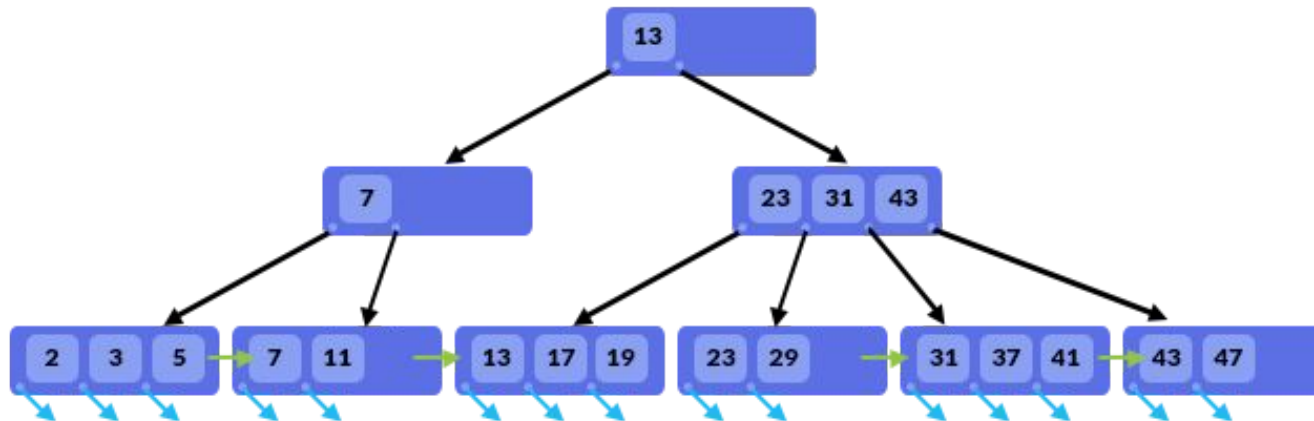


# INDEX : B+ TREE

B+ tree index is a tree structure consisting of Inner nodes and Leaf nodes.

Inner nodes store only value, leaf nodes also stores pointers to row.

B+ tree stores all keys in sorted order.

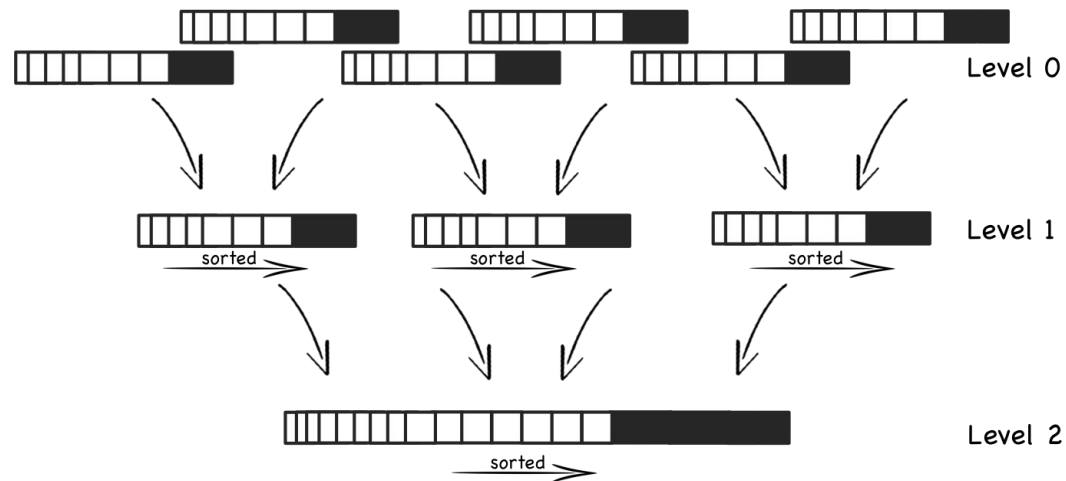


# INDEX : LSM TREE

LSM tree consists of two or more tree-like component data structures.

The top level resides in memory and lower levels resides in disk or other cheap storage.

LSM tree uses combination of hash table and Sorted String Table.



Compaction continues creating fewer, larger and larger files

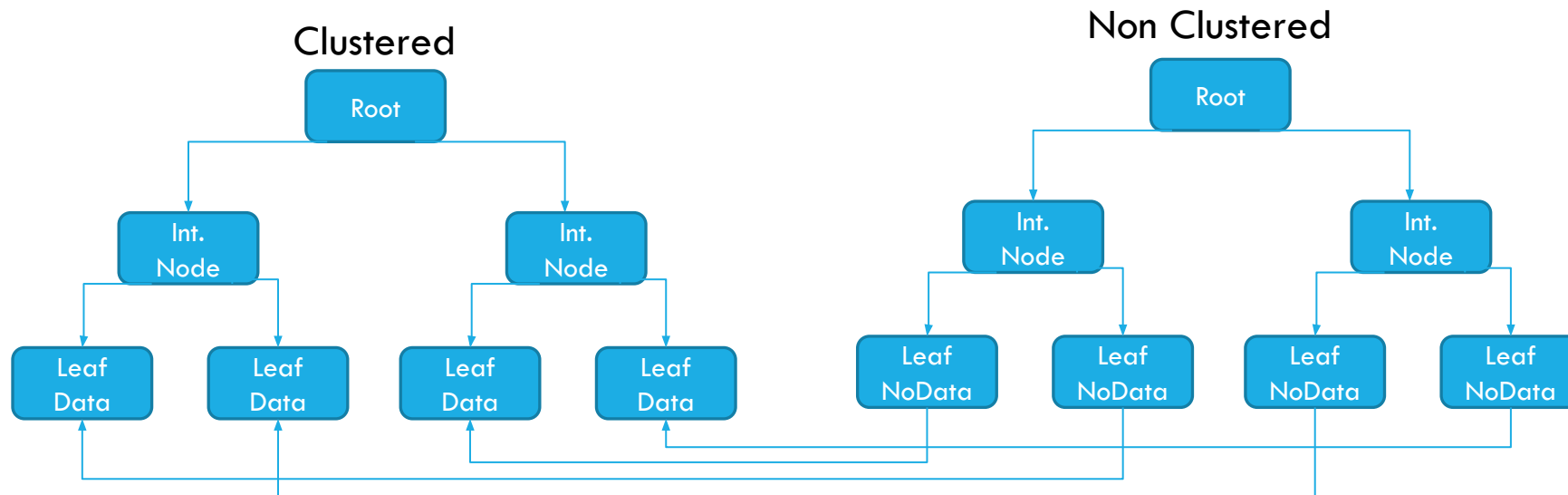
# INDEX : CLUSTERED / NON-CLUSTERED

Clustered index is used to store the table data.

Only one column can be used as clustered index.

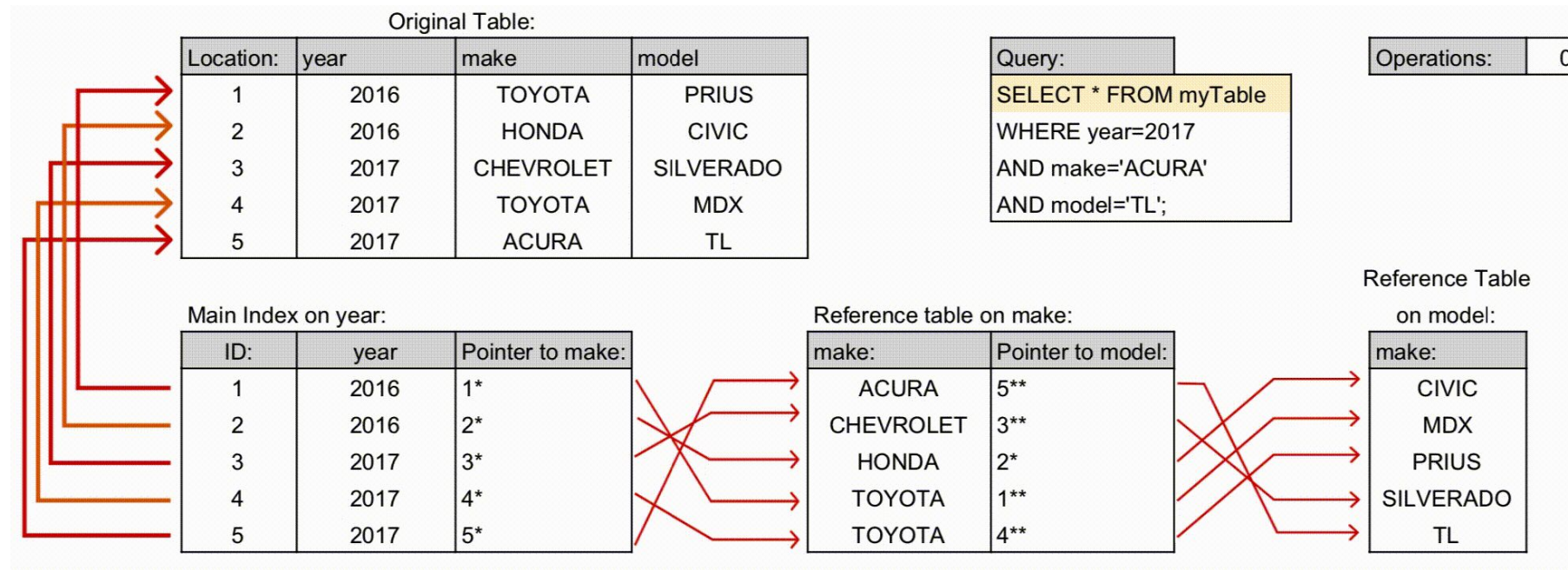
Non-clustered index stores the keys separately and contains pointers to actual records.

A table can have several non-clustered index.



# INDEX : MULTICOLUMN

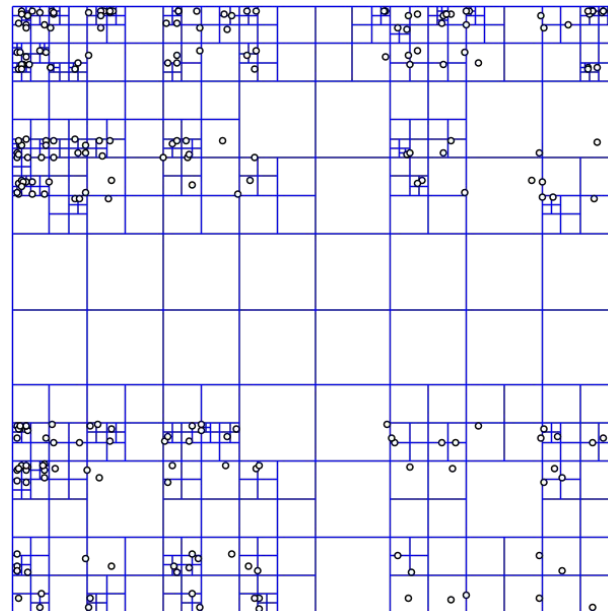
Multi-column indexes use multiple columns in specified order as a key for sorted structure.



# INDEX : QUADTREE

A quad-tree divides range of information in four groups.

Quad-tree uses two columns (criteria) to decide the block containing details of required query range.



Location data with bucket size = 1

# CACHE

If the data is available at a location supporting quick random read and write, performing an operation is extremely quick!

Ability to quickly operate on data:

L1 Cache >> L2 Cache >> L3 Cache >> Memory >> Disk

However, due to limited cache size, answer to following question determines successful utilization of cache.

- What data should be cached ?
- Cache eviction policy
- When should an updated entry written back to database ?

# QUERY

Query flow

Query Optimization

- Data Statistics

- Materialized View

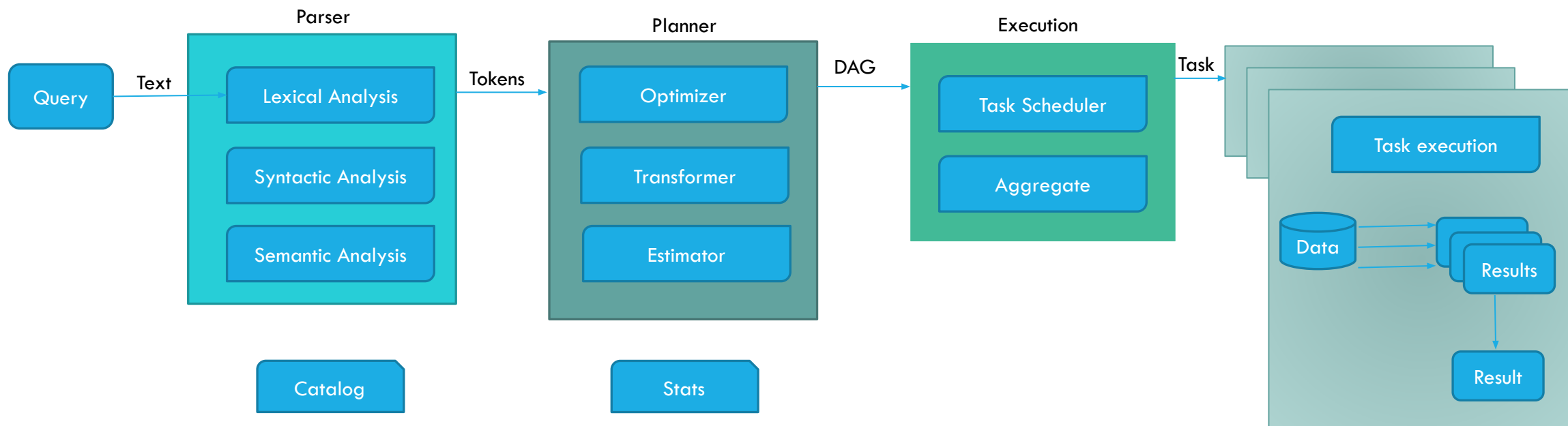
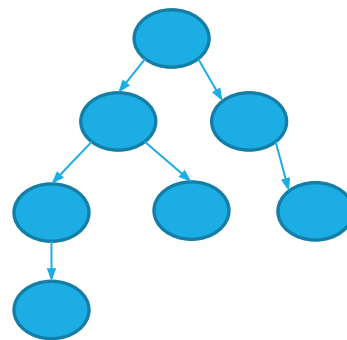
- Execution Engine

  - Parallelized execution

Federated Query



# QUERY FLOW



# QUERY OPTIMISATION

The various system involved in processing a query can be optimised (or tuned) to enhance performance in specific scenario.

Query optimisation can result from Hardware, Database Structure and Database algorithms tuning.

Hardware factors like word-size, hardware layout, size of RAM, CPU cache, speed of processors affect queries performance.

Software components like compression, encoding, indexing, table normalization also equally contribute in query performance.

Apart from these, techniques like cost-based query planning, materialized views, parallelization of sub-tasks during query execution also play a crucial role in query performance.

There are many ways of writing a query to get same result. This also means that a query can be transformed or restructured to get same result, but in a more performant manner.

# QUERY OPTIMIZATION : COST BASED

Cost-based optimization make use of various statistics to decide on appropriate query plan.

e.g. Consider following table stores details of entire India population.

Aadhar	Name	State	Phone	DateOfBirth
1234-5678-90	Ajit Patel	Gujarat	987635421	12-04-1976
2314-5768-07	Neeraj Sharma	Haryana	879038282	30-12-1987

Let's further consider that there's an index on column "Name" as well as "State".

When a query such a following comes:

Select distinct DateOfBirth from POPULATION\_DATA where Name = "XYZ" and State Like "G%"

# QUERY OPTIMIZATION : COST BASED

Select distinct DateOfBirth from POPULATION\_DATA where Name = "XYZ" and State Like "G%"

The Optimizer can choose to filter either on Name or State column first. A cost-based optimizer would evaluate cost for both filters and then choose to apply one (with minimal cost).

Case 1: If Name doesn't exist, it could be efficient to filter on Name column first.

Case 2: If Count(records) with given name > Count(records) for given state, it could be efficient to filter on State column first.

Aadhar	Name	State	Phone	DateOfBirth
1234-5678-90	Ajit Patel	Gujarat	987635421	12-04-1976
2314-5768-07	Neeraj Sharma	Haryana	879038282	30-12-1987

# QUERY OPTIMIZATION : MATERIALIZED VIEWS

There are some queries which is executed very often but they can also be very time-consuming.

E.g. Consider a sales table with many entries.

Product	Sales Date	Profit
Xbox	2022-09-01	30000
Surface Pro	2021-10-09	20000

Month-Year	Total Profit
10-2021	20000
09-2022	30000

Queries such as - Total Profit on monthly basis could be of huge interest. It is possible that newer entries are constantly being added to table and users always want to see latest data.

Even when the table is partitioned on basis of month(Sales Date), there could still be millions of records to perform computation to fetch details of latest month. For previous month, Total Profit could be stored in an archival table.

If a materialized view is constructed on this table to get Total Profit on monthly basis, in case of update to existing record, the corresponding record in materialized view would get updated automatically. This materialized view can serve Total Profit query almost instantly as there is no more any need to perform any table scan during execution of query.

# QUERY OPTIMIZATION : MISCELLANEOUS

Avoid selecting all columns

- ❑ Select \* is in-efficient as it needs to scan all columns
- ❑ Select colA, colB,...colN scans through required columns only and can be extremely performant in columnar databases selecting few columns

Filter data before Join

- ❑ Consider a query: Select Person.id, Person.name, Person.State from Person inner join Address on Person.id = Address.id where Person.name like 'Ami%' and Address.State like 'Guj%'
- ❑ The number of records joined with filters applied later would be more compared to filtering tables first and then applying join (Assuming query planner was unable to perform predicate pushdown)

DISTINCT, UNION, Group By – statements requiring unique selection can take a performance hit. Try to reduce their usage in query

# QUERY OPTIMIZATION : PARALLELIZATION

A query is converted into several tasks.

Several tasks can be executed parallelly on different system.

- E.g. Select Person.id, Person.name, Person.State from Person inner join Address on Person.id = Address.id where Person.name like 'Ami%' and Address.State like 'Guj%'
- Person table and Address table can be scanned in parallel

A task can be further broken down into sub-task.

Each sub-task can be executed in parallel on same system using vectorization.

- E.g. Person table's complete range can be broken down into several ranges. Each thread can independently work on range assigned to it to find Person like 'Ami%'

# QUERY : FEDERATION

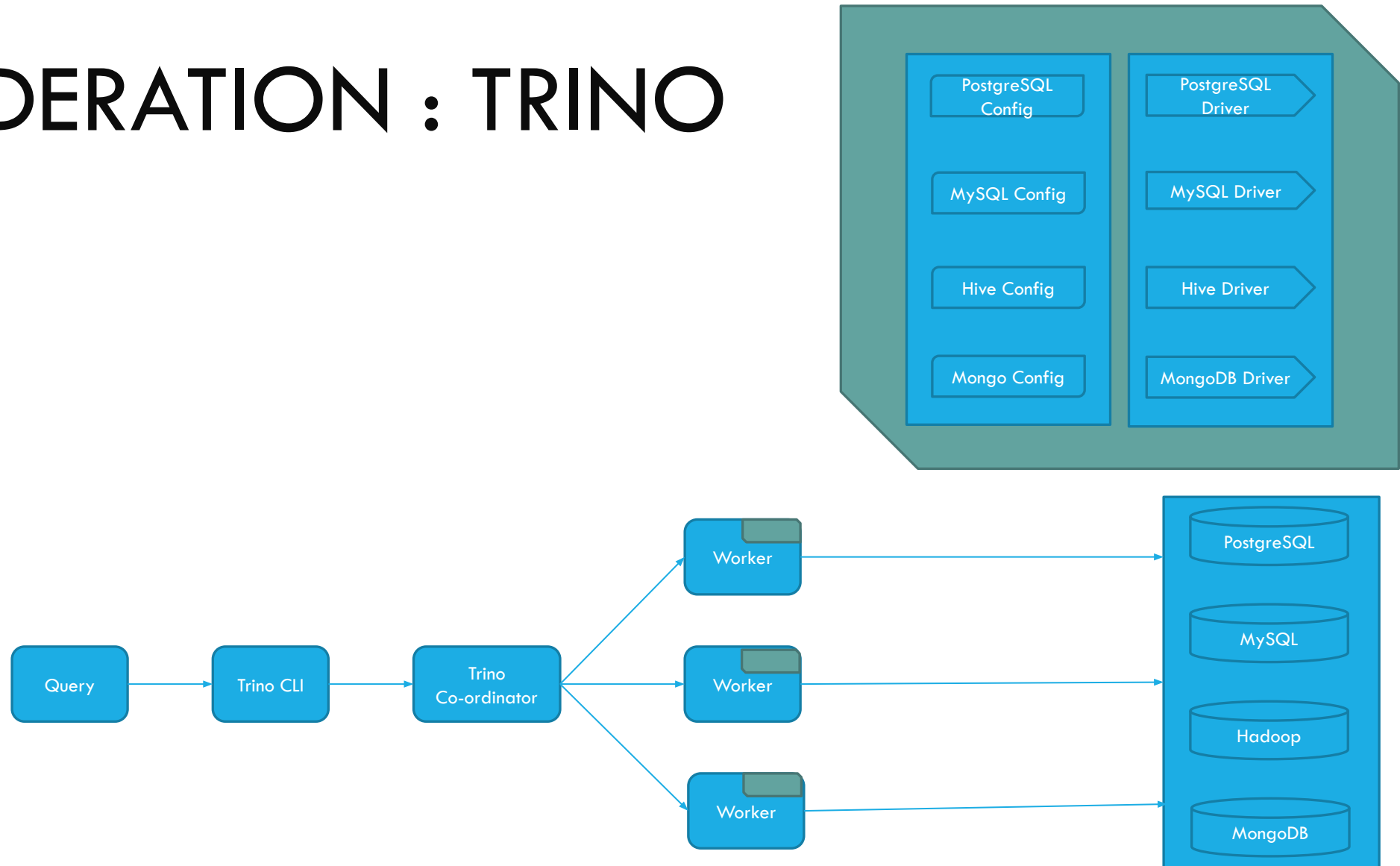
Is a single database good for all use-case ?

Is there a way to query data from multiple system conveniently ?

Is there a way to query different database in same query and combine sub-results to get final result ?



# FEDERATION : TRINO



# DATABASE : CONCLUSION

While choosing a database, the features required by system should be compared against features available in database. The effort and cost to add additional functionality required by system as well as complexity to setup, run and maintain database should be considered.

Metrics helpful in determining the system requirements:

- ❑ Expected Read/Write ratio
- ❑ Expected amount of data to be stored (GB/TB/PB/..)
- ❑ Data duplication across geographies or standalone Datacentre
- ❑ Strong/Eventual/Weak Consistency
- ❑ Structured/Semi-structured/Unstructured data
- ❑ Ad-hoc queries or planned queries
- ❑ Queries per second for read/write
- ❑ Temporal data requirement / Audit
- ❑ Schema evolution requirement
- ❑ Money available to build/maintain system

Database Features

- ❑ Scalability
- ❑ Availability
- ❑ Consistency
- ❑ Persistence
- ❑ Recoverability
- ❑ Performance
- ❑ Bandwidth
- ❑ Throughput
- ❑ Cost

# DATABASE : CONCLUSION

## □ Scalability

- Can new nodes be added dynamically to increase existing capacity ?
- Can new nodes to be added to distribute load on any existing node ?
- Can a node be removed from system without any down-time ?

## □ Availability

- Can the system perform an action even if some of nodes are unavailable ?
- Can the system be made available across different continents ?

## □ Consistency

- Will the result given by the system be same even if query goes to different node each time ?
- How long would it take for latest write to be seen by all clients ?
- Does system allow dirty-reads ?

## □ Persistence & Recoverability

- Will the system be able to recover all/partial data stored earlier ?
- Can the system recover automatically ?

## □ Bandwidth & Throughput

- Can system queries returning millions of rows ?
- How many concurrent users can be handled by system ?

# DATABASE : CONCLUSION

## □ Performance

- How long would it take to write/read data from system ?

- Transactions per second

  - Short transactions

  - Long running transactions

- Read queries per second

  - Ad-hoc query

  - Materialized Query

  - Query with Joins

- Aggregation Queries

## □ Cost

- Can the system be run on a commodity hardware ?

- How much extra effort would be required to add feature not directly supported ?

- How much cost would be incurred in maintenance and to keep system running ?

# REFERENCES

<https://yetanotherdevblog.com/lsm/>

<https://towardsdatascience.com/understanding-apache-parquet-7197ba6462a9>

<https://www.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/ch04.html>

<https://docs.microsoft.com/en-us/sql/relational-databases/data-compression/page-compression-implementation?view=sql-server-ver16>

<https://www.digitalocean.com/community/tutorials/understanding-database-sharding>

<https://hevodata.com/learn/data-replication-strategy/>

[https://docs.oracle.com/database/121/TGSQL/tgsql\\_optncpt.htm#TGSQL192](https://docs.oracle.com/database/121/TGSQL/tgsql_optncpt.htm#TGSQL192)

<https://dev.mysql.com/doc/refman/8.0/en/optimize-overview.html>

<https://www.vertabelo.com/blog/all-about-indexes-part-2-mysql-index-structure-and-performance/>

[https://en.wikipedia.org/wiki/Log-structured\\_merge-tree](https://en.wikipedia.org/wiki/Log-structured_merge-tree)

<https://en.wikipedia.org/wiki/Quadtree>

<https://medium.baqend.com/nosql-databases-a-survey-and-decision-guidance-ea7823a822d>