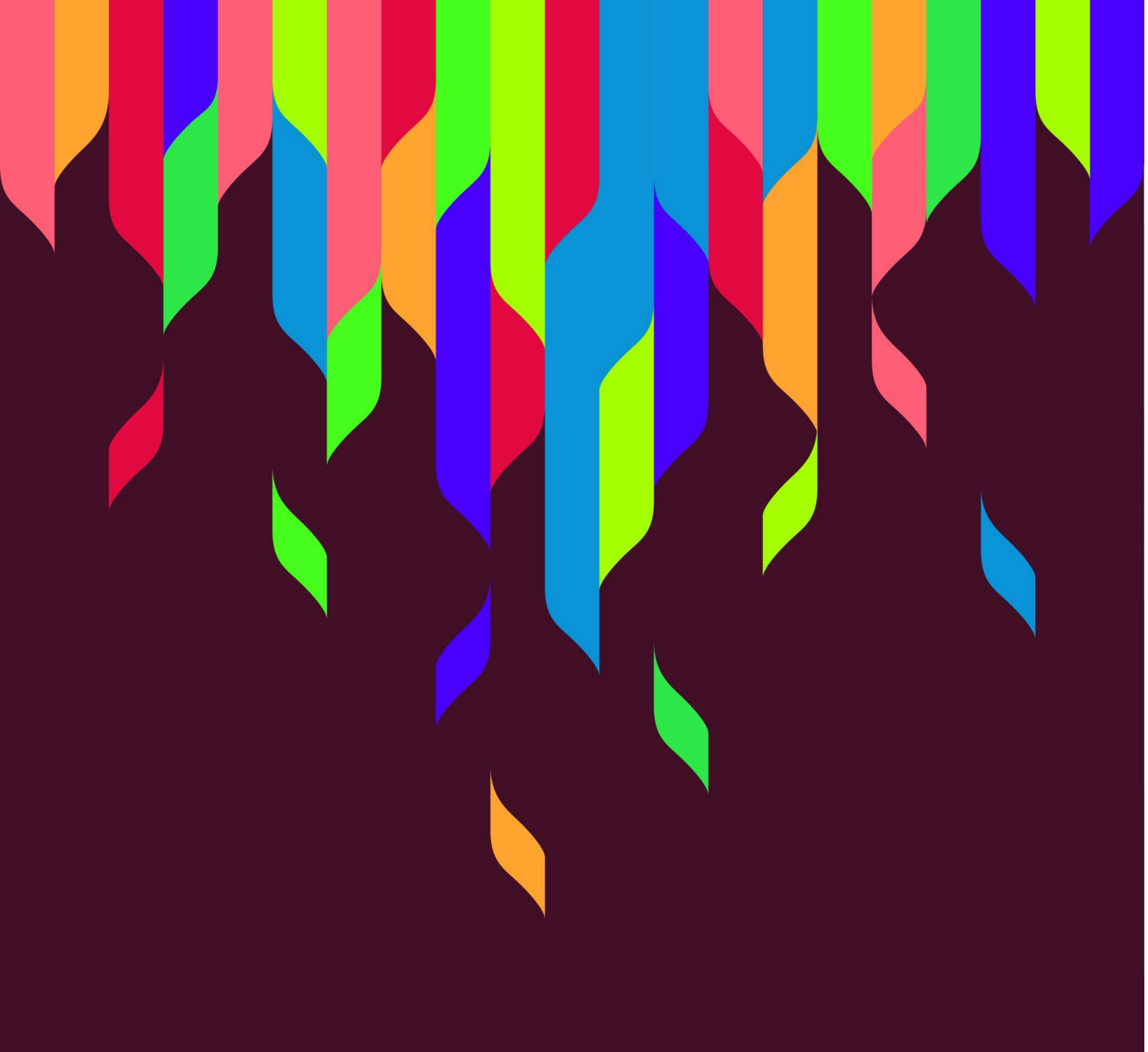
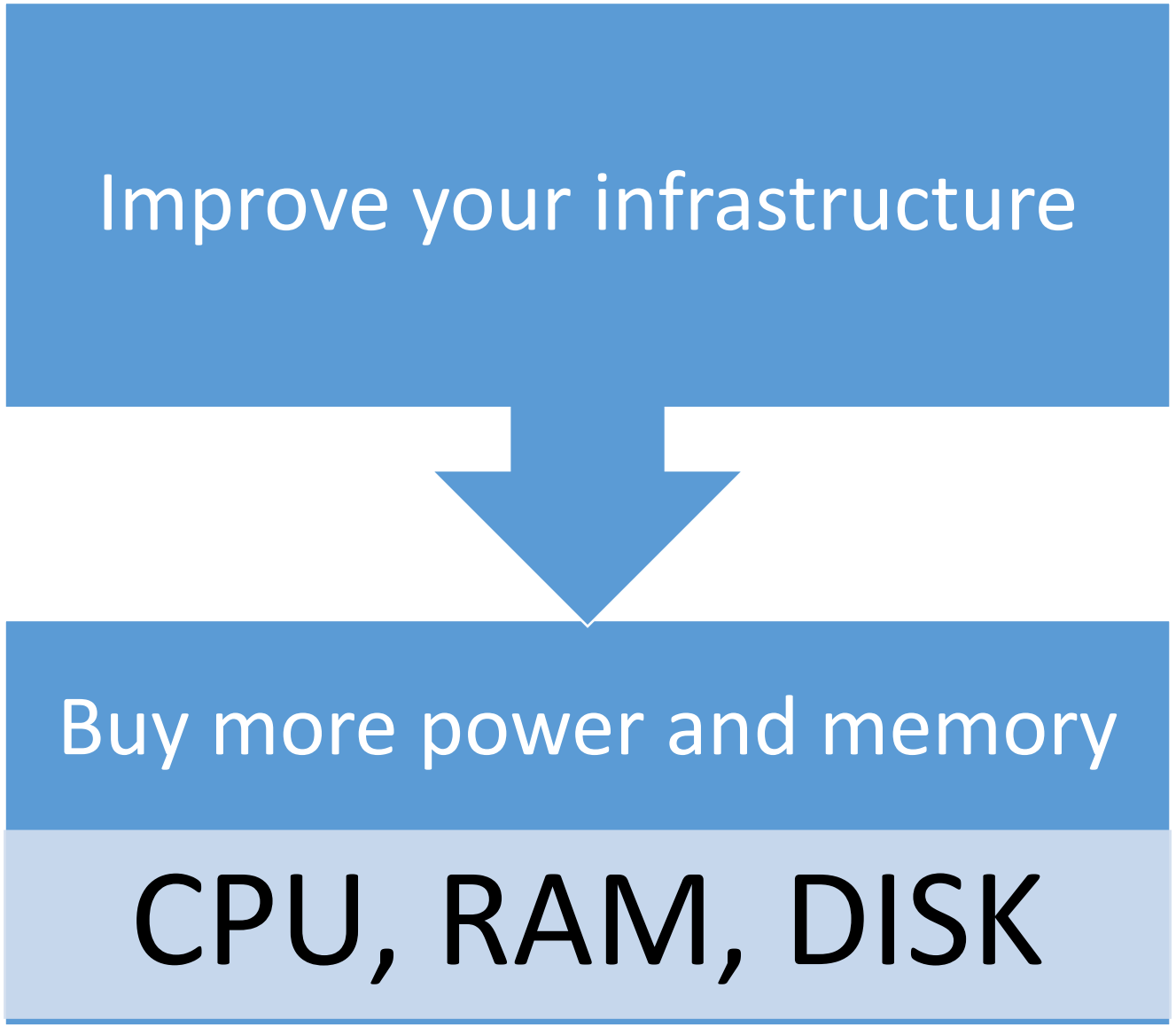


Performance Tuning

Query Optimization

- The overall process of choosing the most efficient means of executing a query
- Determines the most appropriate way by considering query plans

General


Improve your infrastructure
Buy more power and memory
CPU, RAM, DISK

Denormalization

- A database optimization technique
- Objective:
  - To optimize the efficiency of their data store
- How:
  - Systematically add precomputed redundant data to a database to improve the read performance
- Why:
  - Can help avoid costly joins in a relational database made during normalization
- When is it done:
  - Can be done as part of design or delegated to a DBMS

Denormalizaton Techniques

Storing derivable data

- Advantages: no need for lookup when data is needed, no need for calculation every time
- Disadvantages: storage implications, data inconsistency is possible

Pre-joining tables

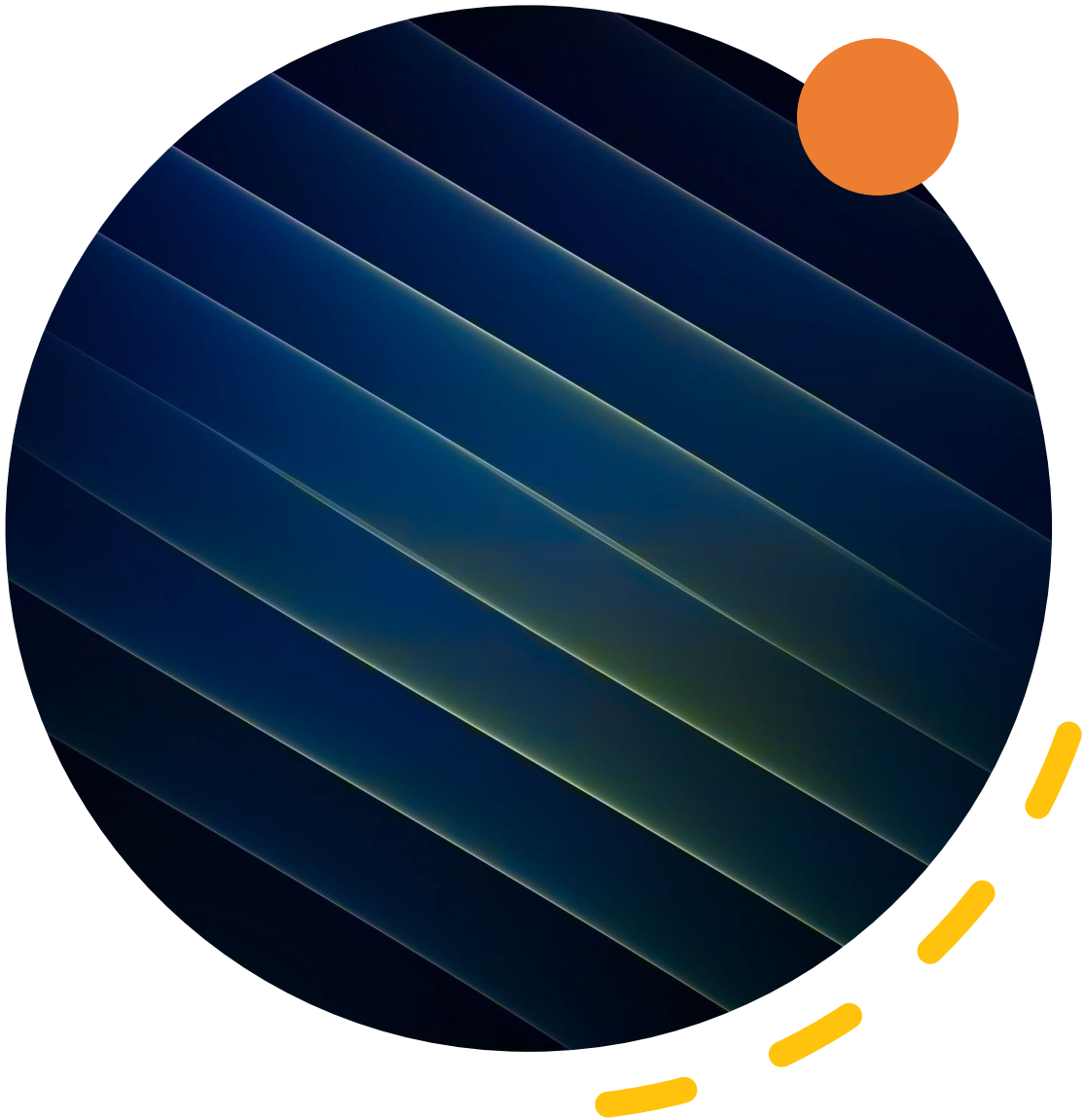
- Advantages: no need for multiple joins
- Dis-advantages: storage implications, additional updates are needed

Keeping details with the master

- Advantages: no need for joins, saves space
- Disadvantages: increased complexity of DML

Adding a short circuit key

- Advantages: fewer tables needed in joins
- Disadvantages: more code needed to maintain consistency

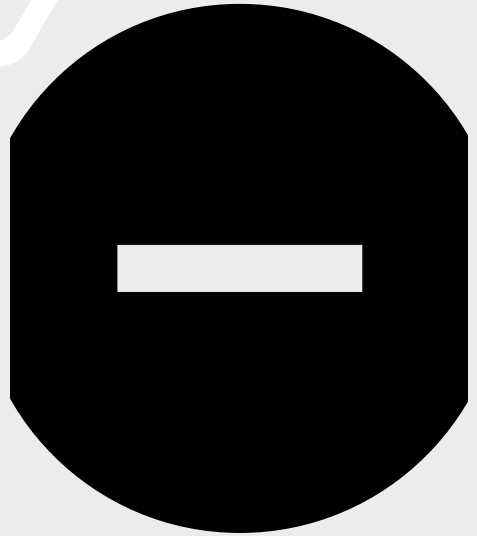


Benefits

- Enhanced query performance
  - JOIN queries are costly on a normalized database
  - Retrieving denormalized data is faster.
- Simpler queries
  - Retrieving data more straightforward because of the smaller number of tables.
- Fewer errors.
  - Working with a smaller number of tables means fewer bugs when retrieving information from a database.
- Easier to manage (in some ways)
  - Fewer tables to backup
  - Need for fewer indexes
- Better support for analytics
  - More suitable for reads

Disadvantages

- Slower and more expensive inserts, updates and deletes
- Leads to inconsistency which has to be managed
- Increases storage requirements
- Sacrifices flexibility



When to use denormalization

- Improving query performance
  - Normalization leads to large number of tables
  - Common queries may therefore involve a lot of joins
  - Reducing number of tables involved can speed things up
- Speeding up reporting/visualization
  - Some calculations/statistics are used frequently.
  - Rather than computing them on live data, precompute them for use in reports/visualization
- Maintaining historical data
  - For example, changes to data are not needed in the live data but may be needed for historical analysis
- Supporting OLAP
  - Complex computations
  - Queries involving live and historical data

SQL (PostgreSQL)

SQL (PostgreSQL)

Find slow queries and add Indexes

- In PostgreSQl the **pg_stats_statements** module tracks execution statistics of SQL statements
- A simple select can identify the queries taking the most time:

```
SELECT *
FROM
  pg_stat_statements
ORDER BY
  total_time DESC;
```

SQL (PostgreSQL)

Find slow queries and add Indexes

- Eliminate Sequential Scans (Seq Scan) by adding indexes (unless table size is small)

SQL (PostgreSQL)

Tune indexes

- If using a multicolumn index, pay attention to order in which you define the included columns
- Try to use indexes that are highly selective on commonly-used data.
  - Narrow the search
  - This will make their use more efficient.

SQL (PostgreSQL)

Remove unused indexes

- When an index exists, it has to be kept updated after every INSERT / UPDATE / DELETE operation.
- Indexes take up space.
- While an index can speed up reading of data (if created properly), it will slow down write operations. This needs to be balanced.
- To find them you need to look at the PostgreSQL statistical counters but these can be reset to zero at times (if a pg_stat_reset() is issued).

```
SELECT tnameAS table_name,
       iname AS index_name,
       i.indisunique,
       idx_scan AS index_scans
FROM   pg_catalog.pg_stat_user_indexes s,
       pg_index i
WHERE  i.indexrelid = s.indexrelid;
```

- If index_scans is 0 or close to 0 then you can drop those indexes but be careful.

SQL (PostgreSQL)

WHERE clause

- Avoid LIKE and ILIKE
  - Fuzzy matching
  - If you need them use appropriate index types (e.g. GIN)

SQL (PostgreSQL)

WHERE clause

- Avoid function calls (or any kind of SQL expression) in WHERE clause
  - This is due to calculation of the cardinality by the optimizer.
  - Example: If we have a table ADDRESSES contains 255691 rows, 37153 of them are addresses in Germany (approx 15% of the rows)
  - If we do a select of addresses for Germany and use UPPER in the where clause

```
SELECT COUNT(*) FROM addresses
 WHERE UPPER(ctr_code) = 'DE';

  COUNT(*)
----------
     37153
```

SQL (PostgreSQL)

WHERE clause

```
-------------------------------------------------------------------------
| Id  | Operation          | Name      | Starts | E-Rows | A-Rows |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |           |      1 |        |      1 |
|   1 |  SORT AGGREGATE    |           |      1 |      1 |      1 |
|*  2 |   TABLE ACCESS FULL| ADDRESSES |      1 |   2557 |  37153 |
-------------------------------------------------------------------------
```

- The cardinality in the execution plan (E-rows) shows the estimated rows calculated by the optimizer.
- If there is a big difference to the number of actual rows, the optimizer was not able to estimate the correct cardinality.
- Here we see that it is 1% of the actual rows – the default the optimizer uses for function calls and expressions.

SQL (PostgreSQL)

WHERE clause

- Functions/Expressions in the WHERE Clause
  - You can try to avoid them
  - Use CHECK constraints
  - Create a function based Index
  - Create a virtual column in your SELECT and use it in the WHERE

SQL (PostgreSQL)

JOINs

- When joining tables, try to use a simple equality statement in the ON clause (i.e. a.id = b.person_id).
- Doing so allows more efficient join techniques to be used (i.e. Hash Join rather than Nested Loop Join)

SQL (PostgreSQL)

JOINs

- Convert subqueries to JOIN statements when possible as this usually allows the optimizer to understand the intent and possibly chose a better plan
- Use EXISTS when checking for existence of rows based on criterion because it “short-circuits” (stops processing when it finds at least one match)

SQL (PostgreSQL)

Order by

- Be aware of nulls
- A default ASC order will always return NULL values at the end of the results .
- If you want to sort potentially NULL strings by descending order but keep all the NULLs at the end? Try the NULLS LAST custom sorting order.

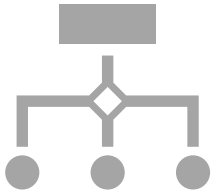
NoSQL

MongoDB

Examine query patterns and profiling

You can enable profiling

db.setProfilingLevel(1)



After your application has run for a while

You can create histograms to show the slowest collections and queries

Showing the number of queries per collection

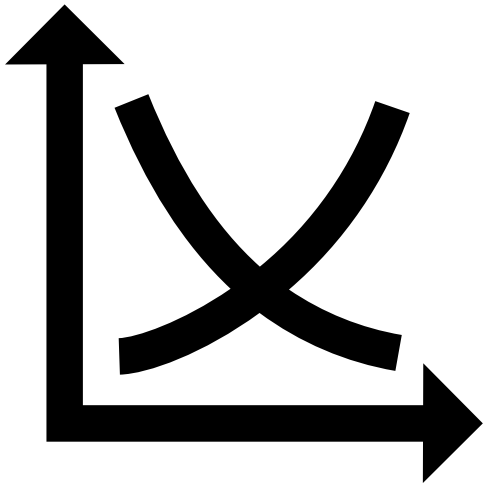
db.system.profile.group({key: {ns: true}, initial: {count: 0}, reduce: function(obj,prev){ prev.count++;}})

Show the slowest collections and queries with time in milliseconds

db.system.profile.group({key: {ns: true}, initial: {millis: 0}, reduce: function(obj, prev){ prev.millis += obj.millis;}})

Find the slowest query

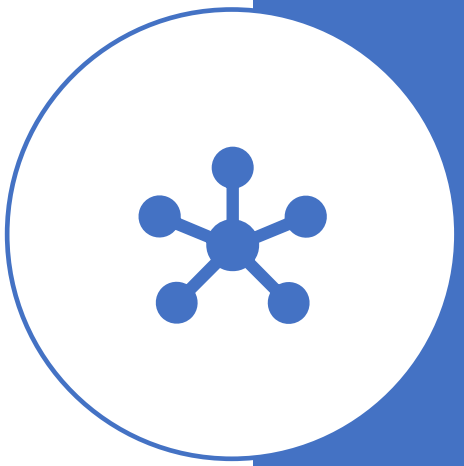
db.system.profile.find().sort({$natural: -



Limit Query Results to Reduce Network Load

- MongoDB cursors return results in groups of multiple documents.
- If you know the number of results you want, you can reduce the demand on network resources by issuing the limit() method.
- Typically used in conjunction with sort operations.
    - E.g. if you need only 10 results from your query to the factmarks collection, you would issue the following command:

```
db.factmarks.find().sort( { marks : -1 }
).limit(10)
```



Store Query Results to Reduce Network Load

- Store the results of frequent sub-queries on documents to reduce read network load

Use Indexes

- Eliminates the need for collection scan
- Covered Query
  - When the query criteria and the projection of a query include only the indexed fields
  - MongoDB returns results directly from the index without scanning any documents or bringing documents into memory.

Use Indexes – Index Intersection

- MongoDB can use an intersection of either the entire index or the index prefix.
  - An index prefix is a subset of a compound index, consisting of one or more keys starting from the beginning of the index

Use Indexes – use hint()

- While the query optimizer usually selects the optimal index for a specific operation you can force MongoDB to use a specific index using the hint() method.
- Use hint() on some queries where you must select a field or field included in several indexes.

- E.g. `db.factmarks.find().hint( { c_name: 1 } )`

Use Indexes Review Index Performance

- Use explain(“executionStats”) on query hinting for each index
- Review the finding

Review Modelling

- While MongoDB has a flexible schema you should always figure out your schema at the beginning of a project so that you won't have to retool everything later on.
- A major advantage of JSON documents is that they allow developers to model data however the application requires.
- Nesting arrays and subdocuments allow you to model complex relationships between data using simple text documents.
- You must make decisions about embedding documents or creating references to documents to manage relationships.

Embedding and Referencing

- Allows you to avoid application joins, which minimizes queries and updates.
- Embedding
  - Data with a 1:1 relationship should be embedded within a single document.
  - Data with a 1:many relationship in which "many" objects appear with or are viewed alongside their parent documents
  - Because these types of data are always accessed together, storing them together in the same document just makes sense.
- Embedded data models allow developers to update related data in a single write operation because single document writes are transactional.

Embedding and Referencing

- Referencing makes more sense when modelling many: many relationships.
- The responsibility for follow-up queries to resolve references lies with the application not the database. May require a lot more calls to the database.
- When to consider referencing:
  - A document is frequently accessed but contains data that is rarely used.
  - Embedding would only increase in-memory requirements, so referencing may make more sense.
  - A part of a document is frequently updated and keeps getting longer, while the remainder of the document is relatively static.
  - The document size exceeds MongoDB's 16MB document limit.
    - Can occur when modelling many:1 relationships, such as product reviews:product, for example.

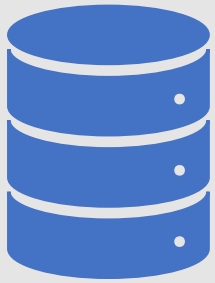
Monitor replication and sharding

- Commands such as rs.status() for replica sets and sh.status() provide a high level status of the cluster.
- Use MongoDB's built-in free monitoring to get information on Operation Execution Times, Memory Usage, CPU Usage, and Operation Counts.
  - mongostat -provides a quick overview of the status of a currently running mongod or mongos (shard) instance
  - mongotop - tracks the amount of time a MongoDB instance spends reading and writing data at a per-collection level
  - Note: you are not required to use these commands for the CA task

NoSQL

Cassandra

Determine the queries first



- In relational databases, the database schema is first created and then queries are created around the DB Schema.
- In Cassandra, schema design should be driven by the application queries.

Choose the correct key

- Choosing the right Row Key (Primary Key) for the column family.
  - The first field in Primary Key is called the Partition Key and all other subsequent fields in primary key are called Clustering Keys.
- The Partition Key is useful for locating the data in the node in a cluster
- The clustering key specifies the sorted order of the data within the selected partition.
- Selecting a proper partition key helps avoid overloading of any one node in a Cassandra cluster.

Duplicate your data

- Encouraged to improve read efficiency

Use Prepared Statements

- Prepared statements get pre-compiled and cached.
- This requires parsing a query only once.
- But it can be executed multiple times with different values.
- See example in python script to load data from PostgreSQL to Cassandra (week 9)

Avoid Allow Filtering

- If possible, avoid ALLOW FILTERING in Cassandra queries
  - This will increase latency impacting performance
- When Allow Filtering is used, Cassandra reads through the data, rather than using an index to seek the data.

Use Indexes

- An index provides a means to access data in Cassandra using a non-partition key.
- You may also reduce or avoid the usage of secondary indexes as they impact read and write performances.

Use Materialized Views

- Use materialized views with automatic updates to make reads faster.
  - E.g. a query for a particular circumstance or group
- Each view is a set of rows which corresponds to rows which are present in the underlying, or base, table specified in the SELECT statement.
- Cannot be directly updated, but updates to the base table will cause corresponding updates in the view.

```
create_materialized_view_statement::=
CREATE MATERIALIZED VIEW [ IF NOT EXISTS
] view_name
AS select_statement
PRIMARY KEY '(' primary_key')’
WITH table_options
```

Bypass Cache

- Use materialized views with automatic updates to make reads faster.
  - E.g. a query for a particular circumstance or group
- Each view is a set of rows which corresponds to rows which are present in the underlying, or base, table specified in the SELECT statement.
- Cannot be directly updated, but updates to the base table will cause corresponding updates in the view.

```
create_materialized_view_statement::=
CREATE MATERIALIZED VIEW [ IF NOT EXISTS
] view_name
AS select_statement
PRIMARY KEY '(' primary_key')’
WITH table_options
```

Querying Patterns in Text

- Break text into trigrams
- Use a GIN index
- Large amount of storage required

- Using an text index and $text operator
- Only one text index allowed per collection
- Doesn’t work with hint
- Doesn’t work with sorting

- Secondary Index
- Custom index
- CONTAINS rather than PREFIX
- Case sensitivity is an issue