



COMS W4121
Computer System for Data Science

Classic Query Processing
and
Fast Query Processing

PROFESSOR:

Eugene Wu

BY:

Yijia Chen (yc3425)
Haotian Zeng (hz2494)
Yiwen Zhang (yz3310)

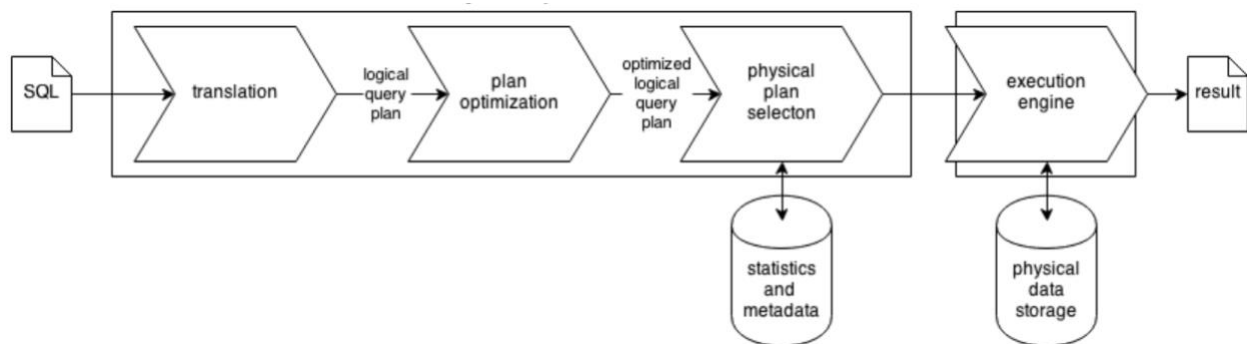
Table of Contents

1. Introduction	3
2. Query Evaluation	5
<i>Logical query plan</i>	7
<i>Physical query plan</i>	7
<i>Push Method: (bottom-up)</i>	8
<i>Pull Method: (top-down)</i>	9
<i>Two Methods Comparison</i>	9
3. Tree Index and Hash Index	11
<i>B-Tree</i>	11
<i>*B+ Tree Indexes (Optional Reading)</i>	13
<i>Hash Table / Hash Indexing</i>	14
4. Partitioning	16
<i>What is Partitioning</i>	16
<i>Partitioning Function</i>	16
5. OLAP (Online analytical processing)	18
<i>Introduction and Application</i>	18
<i>General Principle</i>	19
<i>Queries Examples</i>	19
<i>*Ways to save storage (Optional Reading)</i>	21

1. Introduction

Query processing (as shown in the figure) is the process that transforms a high-level query (SQL) into an equivalent, correct and efficient execution plan expressed in low-level language (such as physical query plan). By execution of the query, users get the desirable results. Such process is executed in Database-Management System (DBMS).

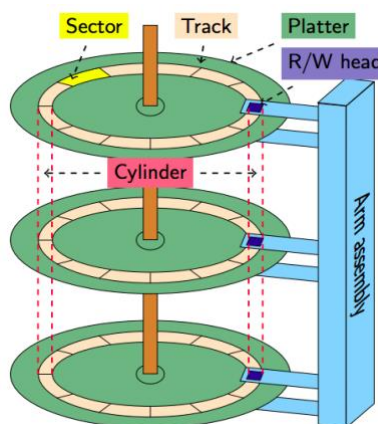
Note:



Data is stored and retrieved in unites called disk blocks or pages. The time to retrieve a disk page varies depending upon location on disk, therefore, relative placement of pages on disk has major impact on DBMS performance.

Generally speaking the seek time is slow, while reading requires a little less time than writing.

Hard drive is one of the most common data storage device that uses magnetic



storage to store and retrieve information, which consists of a number of spinning

platters and an arm assembly with one R/W head for each surface. A surface is one side of a platter. The structure is shown in the picture above.

Compared with Dynamic Random-Access Memory (DRAM), hard drive has its merits and drawbacks:

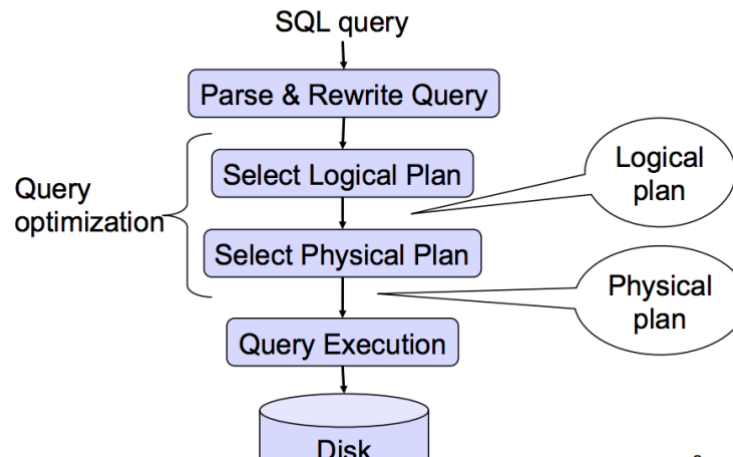
Advantages: *Storage is permanent, necessary for nonvolatile storage of databases.*

Disadvantages: *Hard disks are much slower than DRAM. For continuous throughput, DRAM is typically at least 100 times as fast as a hard disk.*

2. Query Evaluation

A four-step process

Query Evaluation Steps



Step 1: Query rewrite

As base tables in pages may contain large amount of data, then it can be expensive and time-consuming to compute the required aggregates or to compute joins between these tables. In such cases, queries can take minutes or even hours.

As a result, sometimes you might store pre-computed results which can be used to quickly answer queries. To do so, the user's query needs to be rewritten to use the pre-computed results. These pre-computed results are often called materialized views. Materialization means that it has been computed and stored in memory or on disk.

This extremely powerful process is called **query rewrite**, which is employed to quickly answer the query using materialized views.

Step 2: Query parsing

If query rewrite is regarded as preparation, then the real first step of query evaluation is **Query parsing**. Given any query strings, it may contain errors such as syntactically

mistakes (e.g. commas out of place) or semantically mistakes, so that query parsing is needed to turn the **query string** into an algebraic-expression **query plan**.

The process includes:

-
- *Validated for syntax compliance.*
 - *Validated against the data dictionary to ensure that table names and column names are correct.*
 - *Validated against the data dictionary to ensure that the user has proper access rights.*
 - *Analyzed and decomposed into more atomic components.*

Step 3: Query optimization

Find an efficient query plan for executing the query

Our query plan may not born be efficient, so query optimizer is needed to transform initial query plan into a fully equivalent but more efficient one. The database has many internal statistics and tools at its disposal, thus optimizer is usually in a better position than the user to determine the optimal method of statement execution.

In the process, query optimizer first performs straightforward optimizations (*improvements that always result in better performance, like simplifying $5*10$ into 50*).

Next it considers different "query plans" which may have different optimizations, estimates the cost (CPU and time) of each query plan based on the number of rows in the relevant tables, and picks the optimal plan then passes it on to the execution step.

The general process includes:

-
- *Convert SQL query to a **logical query plan** (relational algebra expression).*
 - *Improve the logical query plan and then convert to a **physical query plan**.*

Logical query plan

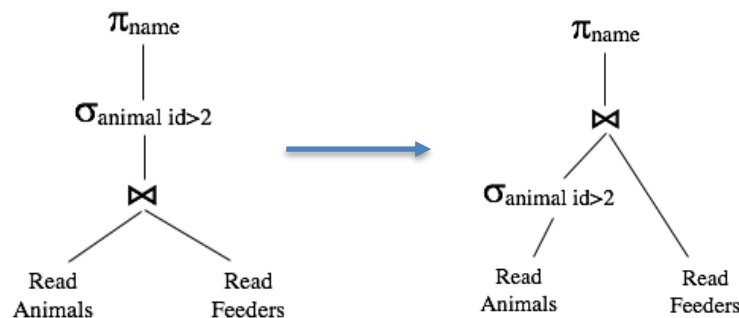
A logical query plan is an extended relational algebra tree. It uses logical operators to connect execute queries. This is created in advance very quickly and represents a first approximation of the query plan that will be executed. Below is the list of logical operators:

Extended Algebra Operators

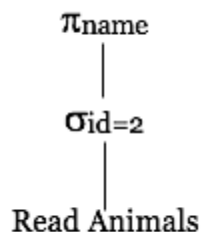
- Union \cup , intersection \cap , difference $-$
- Selection σ
- Projection π
- Join \bowtie
- Duplicate elimination δ
- Grouping and aggregation γ
- Sorting τ
- Rename ρ

Example- Query:

1. “Select animals’ names and their feeders from Dataset ‘animals’ where animal id>2” (before and after optimization)



2. “Select animals’ names from Dataset ‘animals’ where animal id=2”



Physical query plan

Physical Query Plan is the expression in Relational algebra with each operator node assigned some physical algorithms. It does not use exactly the same syntax and operators as the logical query plan; the two plans are different representations of the same algorithm.

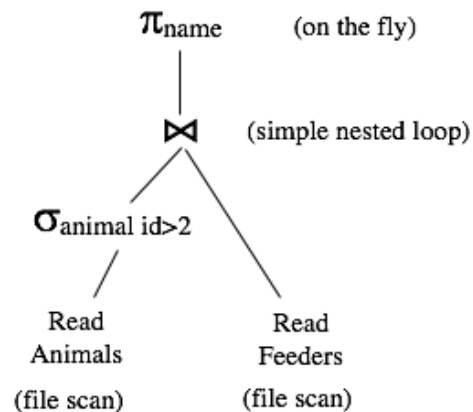
Converting logical query plan to physical query plan should select appropriate algorithms and order of execution.

Physical Query Plan consists of:

Physical Operators (= algorithms used to execute some relational algebra operation, e.g., one-pass join, index-join).

The order in which the physical operations are performed (a tree).

Example- Query(use example in logical query 1):



“Select feeder names from Dataset ‘animals’ and ‘feeders’ where animal id>2”.
Here we convert logical query into physical query.

Step 4: Query execution

As its name suggests, query execution runs each operator in the physical plan to generate the results. There are two ways databases could execute: push and pull.

Push Method: (bottom-up)

Systems do required processing by "pushing" the data that is in the database to render the results for users. For example, in mobile application design, the push pattern sends content to devices automatically, eliminating the dependency on web app servers for pulling content updates.

Example:

“Select animal names from Dataset ‘animals’ where animal id=2”

Define: $t.get("id") = 2$

For page in table animals:

For t in page:

If predicate on t is true:

Add t to output

If output is false/the page is full:

Push to clients/users;

Pull Method: (top-down)

It is a more traditional method of acquiring data. Usually users make requests to the system and pull out the results they would like from database. For example, a web browser requests web pages from web servers.

Example:

$Q = \text{execute}(\text{select}....)$

$Q.\text{next tuple}()$:

While ($Q.\text{hasnext}()$):

$Q.\text{nexttuple}()$

Two Methods Comparison

Reasons for people to use ***push method*** when:

- They require data to be moved and processed in real-time. An example of this would be real-time address confirmation and cleansing from an ecommerce web portal through web service request to the Data Services engine.*
- Need real time interactions with the data.*
- Data is typically being processing from a single primary source.*

Pull method is used where:

- Data latency is less of throughput.*

- *The extracts are driven from Data Services through a batch process.*
- *High volume bulk data.*
- *Data is integrated from many sources providing complex cross system integration.*
- *Change data identification needs to happen within Data Services.*

3. Tree Index and Hash Index

Indexing is used for faster lookup for results. Instead of scanning the entire table for the results, we can reduce the number of page fetches by using ***index structures*** such as ***B-Trees*** or ***Hash Indexes*** to get to the data faster.

An index consists of a ***key*** and ***key values***. A key is the column name of an indexed column. The values in the column are called the key values. Creating an index for a column that will be used as the basis for retrievals from the table will improve the table's retrieval performance.

Understanding the B-trees and hash indexes can help to predict how different queries perform on different storage engines that use these data structures in their indexes, particularly for the reason that lets you choose B-tree or hash indexes in different uses.

B-Tree

Characteristics

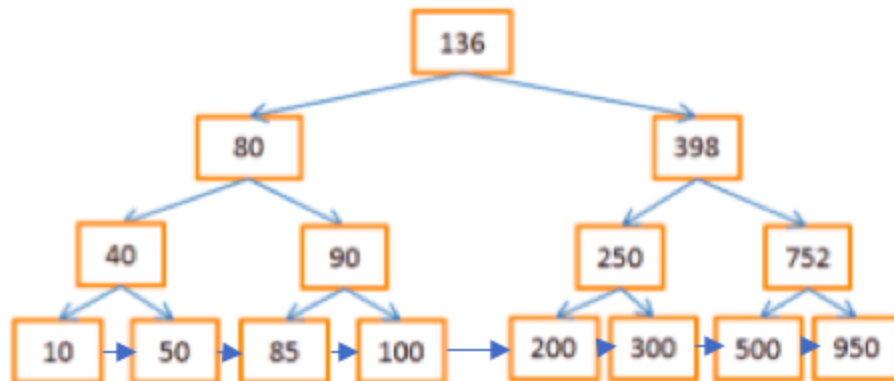
- *B-Tree Indexes are suitable for range queries / range scans since the keys are ordered.*
- *B-Tree Indexes are also suitable point lookups. We can use B-Tree indexes for point lookups but hash indexes are usually more efficient for such workloads.*

Example

A binary search tree is a binary tree with a special property, the key in each node must be (as shown in the figure):

- *Greater than all keys stored in the left sub-tree.*
- *Smaller than all keys stored in the right sub-tree.*

Binary Search Tree



This tree has $N = 15$ elements. Let's say looking for 208:

- I start with the root whose key is 136. Since $136 < 208$, I look at the right sub-tree of the node 136.
- $398 > 208$ so, I look at the left sub-tree of the node 398.
- $250 > 208$ so, I look at the left sub-tree of the node 250.
- $200 < 208$ so, I look at the right sub-tree of the node 200. But 200 doesn't have a right subtree, **the value doesn't exist** (because if it did exist it would be in the right subtree of 200).

Now let's say looking for 40:

- I start with the root whose key is 136. Since $136 > 40$, I look at the left sub-tree of the node 136.
- $80 > 40$ so, I look at the left sub-tree of the node 80.
- $40 = 40$, **the node exists**. I extract the id of the row inside the node (it's not in the figure) and look at the table for the given row id.
- Knowing the row id let me know where the data is precisely on the table and therefore I can get it instantly.

The **cost of the search is $\log(N)$** . We will learn fancier versions in **W4111 Introduction to Databases** class!

*B+ Tree Indexes (Optional Reading)

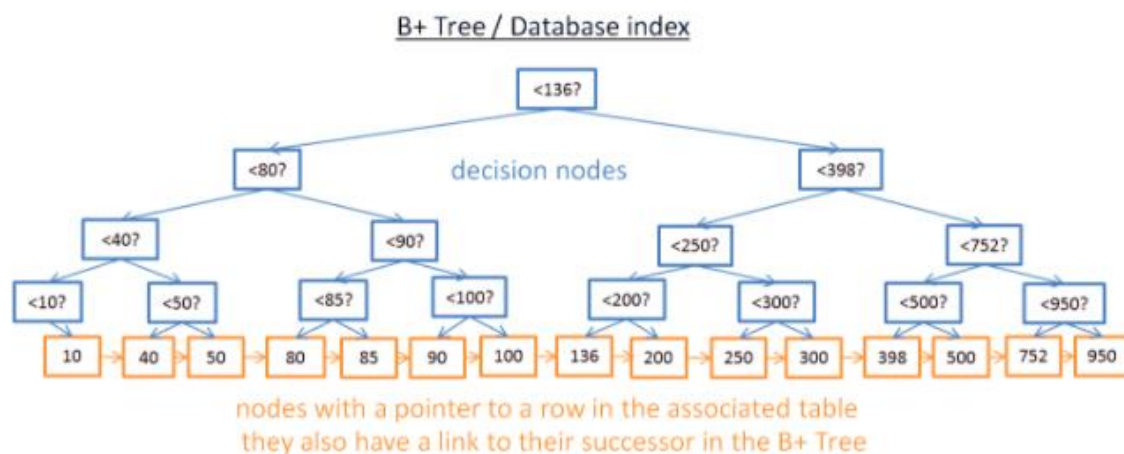
Characteristics

Although this tree works well to get a specific value, there is a BIG problem when we need to **get multiple elements between two values**. It will cost $O(N)$ because we'll have to look at each node in the tree and check if it's between these 2 values (for example, with an in-order traversal of the tree). Moreover this operation is not disk I/O friendly since we'll have to read the full tree. We need to find a way to efficiently do a **range query**. To answer this problem, modern databases use a modified version of the previous tree called B+ Tree.

Example

Modern databases use a modified version of the previous tree called B+ Tree. In a B+ Tree:

- Only the lowest nodes (the leaves) **store information** (the location of the rows in the associated table).
- The other nodes are just here **to route to the right node during the search**.



With this B+ Tree, if look for values between 40 and 100:

- Look for 40 (or the closest value after 40 if 40 doesn't exist).
- Gather the successors of 40 using the direct links to the successors until reach 100.

Advantage

Let's say we found M successors and the tree has N nodes. The search for a specific node costs $\log(N)$ like the previous tree. But, once we have this node, we get the M successors in M operations with the links to their successors. **This search only costs $M + \log(N)$ operations** vs N operations with the previous tree. Moreover, we don't need to read the full tree (just $M + \log(N)$ nodes), which means less disk usage. If M is low (like 200 rows) and N large (1 000 000 rows) it makes a big difference.

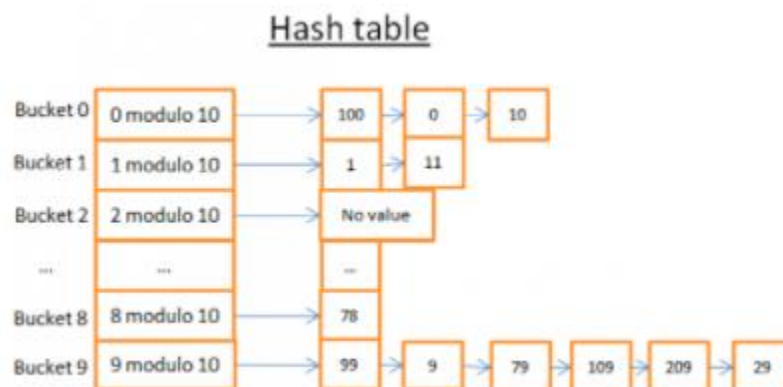
Hash Table / Hash Indexing

Characteristics

The hash table is a data structure that quickly finds an element with its key. To build a hash table we need to define:

- *A key for your elements.*
- *A hash function for the keys. The computed hashes of the keys give the locations of the elements (called **buckets**).*
- *A function to compare the keys. Once we found the right bucket we have to find the element we're looking for inside the bucket using this comparison.*

Simple Example



This hash table has 10 buckets. The Hash function used here is the modulo 10 of the key. In other words we only keep the last digit of the key of an element to find its bucket:

- *If the last digit is 0 the element ends up in the bucket 0,*
- *If the last digit is 1 the element ends up in the bucket 1,*
- *If the last digit is 2 the element ends up in the bucket 2,*
- ...

The compare function used is simply the equality between 2 integers.

Let's say we want to get the element 78:

- *The hash table computes the hash code for 78 which is 8.*
- *It looks in the bucket 8, and the first element it finds is 78.*
- *It gives we back the element 78.*
- ***The search only costs 2 operations*** (1 for computing the hash value and the other for finding the element inside the bucket).

Advantage

Hash Indexes is the general recommendation for **equality based lookups** since we avoid the overhead of going down the B-Tree all the way to the leaf block, and then searching for the desired key within the leaf block and the corresponding row locator.

4. Partitioning

What is Partitioning

Partitioning is a database process where very large tables are divided into multiple partitions. Each partition can be stored on different machines or the same machine. By splitting a large table into smaller and individual tables, queries that access only a fraction of the data can run faster because there is less data to scan.

The main goal of partitioning is to aid in maintenance of large tables and to reduce the overall response time to read and load data for particular SQL operations.

Partitioning Function

Hash partitioning (Use more often): *applies a hash function to some attributes that yield the partition number. This strategy allows exact-match queries on the selection attribute to be processed by exactly one node and all other queries to be processed by all the nodes in parallel.*

List Partitioning (Lookup Table): *a partition is assigned a list of values. If the partitioning key has one of these values, the partition is chosen. For example, all rows where the column Country is either Qatar, UAE, Oman, Kuwait, Bahrain or Saudi Arabia could build a partition for the GCC countries.*

Range partitioning (Used more often): *selects a partition by determining if the partitioning key is inside a certain range. An example could be a partition for all rows where the column zip code has a value between 70000 and 79999. It distributes tuples based on the value intervals (ranges) of some attribute. In addition to supporting exact-match queries (as in hashing), it is well-suited for range queries. For instance, a query with a predicate “A between A1 and A2” may be processed by the only node(s) containing tuples.*

Round robin: *the simplest strategy, it ensures uniform data distribution. With n partitions, the i th tuple in insertion order is assigned to partition $(i \bmod n)$. This strategy enables the sequential access to a relation to be done in parallel. For example, if you have 3 nodes in database partition, then:*

*1st ROW will go to node1
2nd ROW will go to node2
3rd ROW will go to node3
4th ROW will go to node1
5th ROW will go to node2*

... And the process keeps on rotating.

5. OLAP (Online analytical processing)

Which enable users to easily and selectively extract and view data from different points of view

Introduction and Application

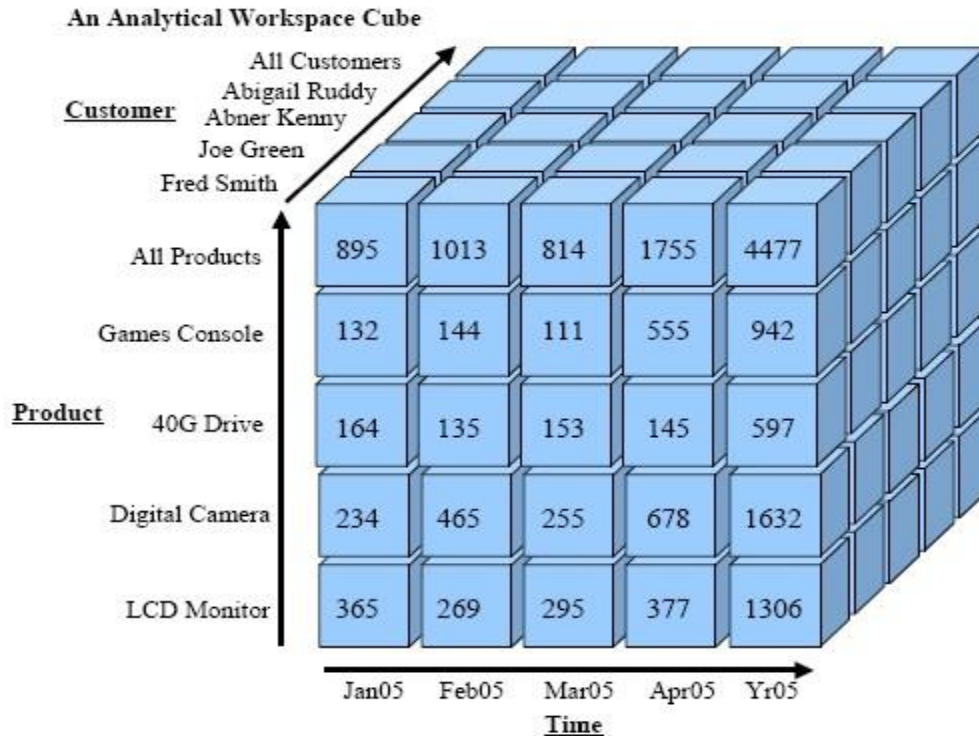
Online Analytical Processing is an approach to answering **multi-dimensional analytical queries** swiftly in computing. Whereas a relational database can be thought of as two-dimensional, an OLAP based database can be multidimensional, which considers each data attribute (such as animal species, cage size, and age) as a separate "dimension."

Traditionally, queries against a data warehouse were created to answer questions like: 'How much profit did each regional division make?' But organizations were quickly realizing that the wealth of data in their warehouse enabled much more interesting and sophisticated reports and analysis to be performed.

Now a typical query might be: 'For calendar quarters this year, show the percentage change in sales for Electronic products sold to customers who live in the USA compared to the same quarters last year'.

This type of query can be difficult to express relationally using SQL, but would be simple to describe multi-dimensionally using the OLAP option. It is often called a multi-dimensional query and business analysts create these queries on-line and 'slice and dice' and refine their view of the data to uncover and highlight underlying trends and hence online analytical processing, or OLAP, was born.

One of the ways to store the multidimensional data types is OLAP Cubes. In the OLAP Cube below, it is defined by three dimensions, Time, Product and Customer, and the cells within the cube are the data values, (which are also known as measures).



General Principle

Bulk writes: *A large amount of data is imported at the same, which would be much quicker than a conventional record-by-record/programmatic insert.*

Few updates: *The process of update is relatively slow, we should make our database as complete as possible to avoid such operation.*

Read mostly: *Though the contents in the database can be changed, the database is designed on the basis that such changes will be very infrequent compared with the number of occasions when the contents are read.*

Query Examples

People usually ask similar questions, but they are slightly different. For example:

Q1: select Month, Sum (Sales)

Q2: select Month, Sum (Sales), where Month = 'Feb'

Q3: select Month, Average (Sales)

Q4: select Month, Region, Sum (Sales)

Q5: select Month, Sum (Sales), where Region = 'US'

Q6: select Year, Sum (Sales)

For Q1 and Q3, we can group by Month and Aggregate Sales for each month, as the following chart shows.

Month	Sales
January	<i>Sum/count/avg</i>
February	<i>Sum/count/avg</i>
March	<i>Sum/count/avg</i>
....

For Q2, we can group by Month and aggregate Sales for each month, then filter for February.

For Q4, we can group by Month and Region, and aggregate sales, as the following chart shows.

Sales by month and region	US	Europe	Asia
January	<i>Sum/count/avg</i>	<i>Sum/count/avg</i>	<i>Sum/count/avg</i>
February	<i>Sum/count/avg</i>	<i>Sum/count/avg</i>	<i>Sum/count/avg</i>
March	<i>Sum/count/avg</i>	<i>Sum/count/avg</i>	<i>Sum/count/avg</i>
....			

For Q5, we can group by Month and Region, and aggregate sales, then filter for US only.

For Q6, we can group by Year and Aggregate Sales for each year.

****Ways to save storage (Optional Reading)***

Use integers, rather than strings.

Storage of strings will take up more space than integers. For example, we can use a serie of binary integers to express 26 letters in English, and thus to store the names which originally marked by letters.

Avoid duplicate integers.

Unavoidably, some integers may appear several times in a sequence. As a result, we can gather them together and rewrite them in the form such as (1, 2) and (2, 3) to substitute 1, 1, 2, 2, 2 where the second number in the bracket indicate the repeat time.

Use the way “+X” in a sequence.

For example, if we want to store the integers 10, 12, 13, 14..... Rather than store them as “10”, “12” directly, which may take up 32-bits space for each integer, we can express 12 in the sequence as “+2”, 13 as “+1”, 14 as “+1”...To compare, we only need 4-bits space for each integer.

Use a column-oriented Database.

A column-oriented database serializes all of the values of a column together, then the values of the next column, and so on. For example, originally 17 attributes need 4GB for storage. As far as we make it into the single attribute in column-oriented database, we only need 1GB. Generally, it is about 240 MB/column on average.