DSC-204A: Scalable Data Systems, Winter 2024

Parallelism Basics - 1

Lecturer: Hao Zhang *Scribe: Sanchit Goel, Xuanyu Chen, Zheyu Fu*

1 Column-oriented Storage and Schemas

1.1 Schemas for Analytics

One of the ways to improve OLTP database for OLAP usage is to change the schema of the database. Here are some popular schemas for storing data:

- Relation (SQL)
- Document (NoSQL)
- Graph (GraphQL)
- Network
- Hierarchy
- Stars
- Snowflake

We will look into Star and Snowflake schemas.

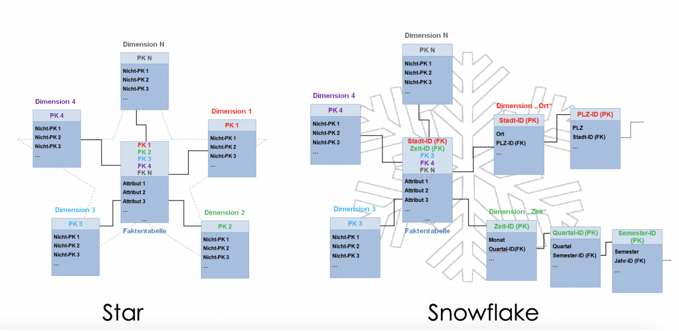
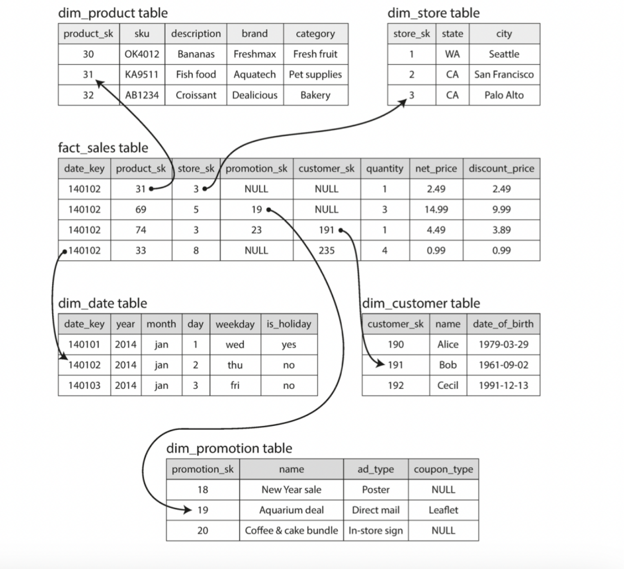


Figure 1: Star and Snowflake Schema

Figure 1 shows the structural differences between the two schemas. The star schema, represented on the left has a central table that stores all the attributes and links to all the sub tables. Similarly, snowflake schema also has a central table that links to sub-tables but the sub-tables can have child tables as well.



dim_product table

product_sk	sku	description	brand	category
30	OK4012	Bananas	Freshmax	Fresh fruit
31	KA9511	Fish food	Aquatech	Pet supplies
32	AB1234	Croissant	Dealicious	Bakery

dim_store table

store_sk	state	city
1	WA	Seattle
2	CA	San Francisco
3	CA	Palo Alto

fact_sales table

date_key	product_sk	store_sk	promotion_sk	customer_sk	quantity	net_price	discount_price
140102	31	3	NULL	NULL	1	2.49	2.49
140102	69	5	19	NULL	3	14.99	9.99
140102	74	3	23	191	1	4.49	3.89
140102	33	8	NULL	235	4	0.99	0.99

dim_date table

date_key	year	month	day	weekday	is_holiday
140101	2014	jan	1	wed	yes
140102	2014	jan	2	thu	no
140103	2014	jan	3	fri	no

dim_customer table

customer_sk	name	date_of_birth
190	Alice	1979-03-29
191	Bob	1961-09-02
192	Cecil	1991-12-13

dim_promotion table

promotion_sk	name	ad_type	coupon_type
18	New Year sale	Poster	NULL
19	Aquarium deal	Direct mail	Leaflet
20	Coffee & cake bundle	In-store sign	NULL

Figure 2: Star Schema Tables

Figure 2 delves into the specifics of the Star Schema, showcasing the tables and their interconnections. The Star Schema is characterized by a central table, referred to as the fact table, storing essential attributes and links to various sub-tables. This schema simplifies the structure and enhances query efficiency for analytics. Let's suppose, you are only interested in the product table for analysis, you only need to query the product table.

Conversely, the Snowflake Schema, while also featuring a central table connecting to sub-tables, introduces a more normalized structure. Notably, the sub-tables in a snowflake schema can have further child tables, increasing the level of normalization and complexity in the schema.

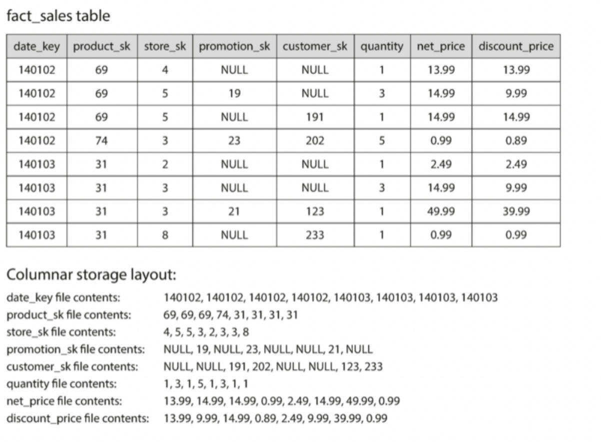
1.2 Column-oriented Storage

The second way of improving OLTP database for OLAP usage is by utilizing column-oriented storage. In traditional row-oriented storage, data is stored sequentially row by row on disk. Analytical queries, especially those typical in OLAP usage, often require scanning large amounts of historical data and performing statistical or complex machine learning functions. The inefficiency arises from the need for random access to various rows, leading to slower query execution.

Column-oriented storage, on the other hand, addresses this inefficiency by reorganizing data values based on columns rather than rows. The advantages become apparent when considering the typical workload characteristics of OLAP. Analytical queries, prevalent in OLAP workloads, witness significant speed enhancements, with query execution speeds potentially accelerating up to 100 times.

1.2.1 Implementation of Column-Oriented Storage

The implementation of column-oriented storage involves a fundamental rearrangement of the traditional row-oriented storage model. Rather than storing data row by row, each column is stored consecutively as shown in figure 3, allowing for streamlined access to specific columns during query execution.



fact_sales table

date_key	product_sk	store_sk	promotion_sk	customer_sk	quantity	net_price	discount_price
140102	69	4	NULL	NULL	1	13.99	13.99
140102	69	5	19	NULL	3	14.99	9.99
140102	69	5	NULL	191	1	14.99	14.99
140102	74	3	23	202	5	0.99	0.89
140103	31	2	NULL	NULL	1	2.49	2.49
140103	31	3	NULL	NULL	3	14.99	9.99
140103	31	3	21	123	1	49.99	39.99
140103	31	8	NULL	233	1	0.99	0.99

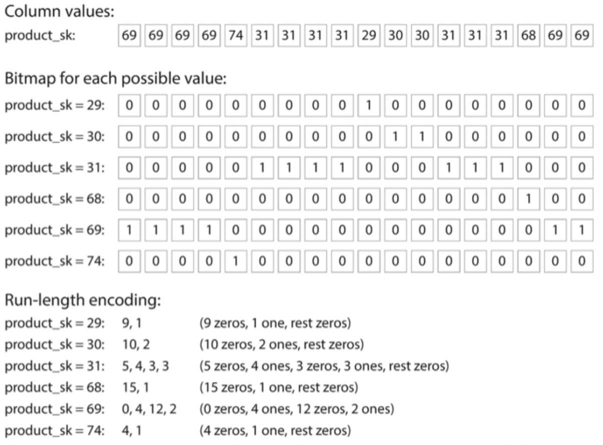
Columnar storage layout:

date_key file contents:	140102, 140102, 140102, 140102, 140103, 140103, 140103, 140103
product_sk file contents:	69, 69, 69, 74, 31, 31, 31, 31
store_sk file contents:	4, 5, 5, 3, 2, 3, 3, 8
promotion_sk file contents:	NULL, 19, NULL, 23, NULL, NULL, 21, NULL
customer_sk file contents:	NULL, NULL, 191, 202, NULL, NULL, 123, 233
quantity file contents:	1, 3, 1, 5, 1, 3, 1, 1
net_price file contents:	13.99, 14.99, 14.99, 0.99, 2.49, 14.99, 49.99, 0.99
discount_price file contents:	13.99, 9.99, 14.99, 0.89, 2.49, 9.99, 39.99, 0.99

Figure 3: Star Schema Tables

Column-oriented storage proves to be storage-efficient too. The homogeneous nature of columns allows for effective compression techniques, taking advantage of high repetitiveness within a column. This compression not only saves storage space but also translates into significant cost savings, particularly in cloud-based storage services.

1.2.2 Column Compression



Column values:

product_sk:	69	69	69	69	74	31	31	31	31	29	30	30	31	31	31	68	69	69

Bitmap for each possible value:

product_sk = 29:	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
product_sk = 30:	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0
product_sk = 31:	0	0	0	0	0	1	1	1	1	0	0	0	1	1	1	0	0	0
product_sk = 68:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
product_sk = 69:	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1
product_sk = 74:	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0

Run-length encoding:

product_sk = 29:	9, 1	(9 zeros, 1 one, rest zeros)
product_sk = 30:	10, 2	(10 zeros, 2 ones, rest zeros)
product_sk = 31:	5, 4, 3, 3	(5 zeros, 4 ones, 3 zeros, 3 ones, rest zeros)
product_sk = 68:	15, 1	(15 zeros, 1 one, rest zeros)
product_sk = 69:	0, 4, 12, 2	(0 zeros, 4 ones, 12 zeros, 2 ones)
product_sk = 74:	4, 1	(4 zeros, 1 one, rest zeros)

Figure 4: Star Schema Tables

But how to implement column compression? Data compression in column-oriented storage involves exploiting repetitive patterns within columns. For instance, utilizing a sparse coding technique like bitmap encoding enables the creation of a map indicating which values are equal to one as shown in figure 4, effectively compressing sparse values. This reduces the storage footprint, a critical consideration in cloud environments where data movement incurs costs.

1.3 Summary of Important Concepts

We introduce hashtable indexes, SSTable, LSM, B-tree. And we introduce two important tricks which are self-balanced tree and bloom filter. If you will be reading any cloud data article, the general rule of thumbs are that LSM Tree is better for in-memory database, while B-Tree is better for classic relational and on-disk database. And these are what happening in industries. Then we introduce OLAP versus OLTP. OLTP is still being increasingly hotter and hotter. Finally we introduce data warehouse.

2 Parallelism

2.1 Express Data Processing in Abstraction

The central issue for parallelism is that workload takes too long for one processor. We want to process them faster in given time, so we want to deploy more processors for the processing. The basic principle is that we split up workloads across professors and perhaps also across machine and workers (aka "Divide and Conquer"), like what we do in PA1. As Figure 5 shows, We continue to divide and do the work on the divided partition, and then we merge the result back.

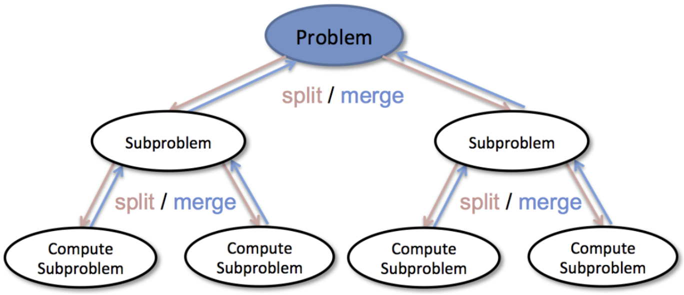


Figure 5: Divide and Conquer

Figure 6 shows the abstraction of data processing. After applying processing functions original data will be transformed into result data which may be in the format of data or ML models. Processing functions could be sum, mean, Page rank, supervised learning, clustering, model inference and so on.

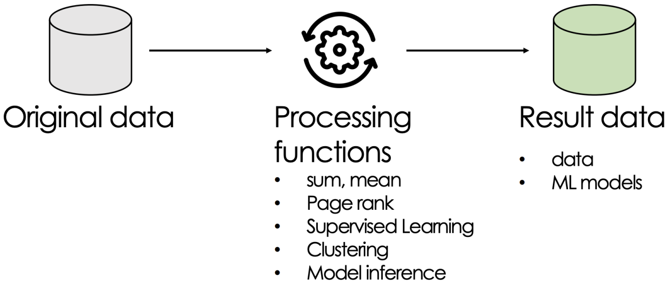


Figure 6: Abstraction of Data Processing

But how to express arbitrarily complex processing function? People normally use dataflow graph, which is common in parallel data processing. A directed graph representation of a program contains:

- Vertices: abstract operations from a restricted set of computational primitives.
- Edges: data flowing directions (hence data dependency).

The dataflow graph enables us to reason about data-intensive programs at a higher level.

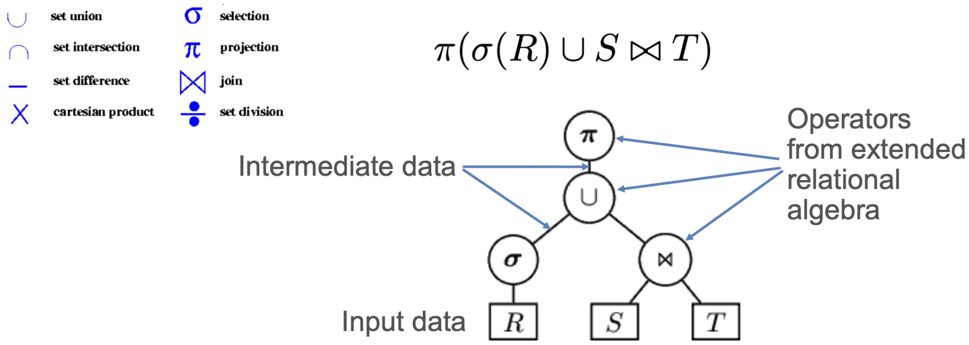


Figure 7: An example of relational dataflow graph.

Figure 7 demostrates an example of relational dataflow graph. The edges are intermediate data, while the nodes are operators from extended relational algebra. We start with R, S, T. And then we have edges that fit them into two operations, one is selection, the other is the join. The result of these two operations will complete a union function, and finally it will goes out to apply another function, projection.

Likewise, Figure 8 demostrates an example of machine learning dataflow graph.

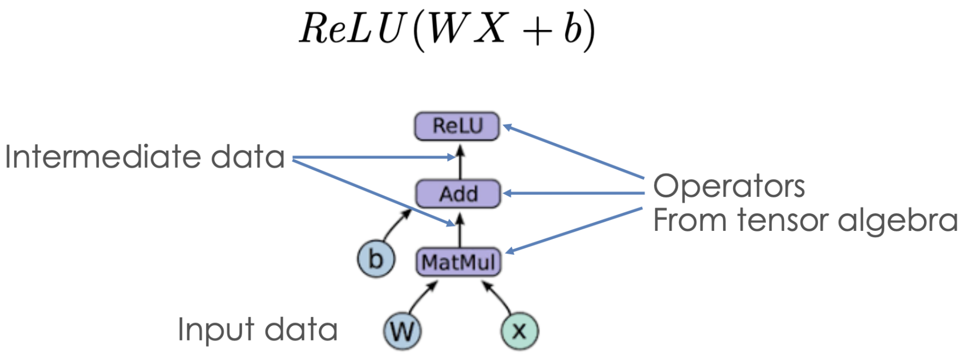


Figure 8: An example of machine learning dataflow graph.

2.2 Parallelisms

When utilizing dataflow graph representation, we gain a structured way of visualizing both data and data processing functions. This representation enables us to exploit various forms of parallelism effectively. The fundamental concept underlying parallelism involves distributing the workload of processing functions across multiple nodes. Each node is responsible for handling a portion of the workload, after which we will synchronize the results to produce the final outcome.

Parallelism can be categorized into several types based on whether we partition the data, the graph, or both. To simplify the classification, we consider three primary options for data: shared, replicated, and partitioned. Additionally, we can choose to replicate or partition the functions (i.e., programs). For the purpose of elucidating key parallelism paradigms, we will not delve into scenarios involving significantly large functions, such as ChatGPT.

data func	Shared / Replicated	Partitioned
Replicated	N/A (rare cases)	Data parallelism
Partitioned	Task parallelism	Hybrid parallelism

Table 1: Types of Parallelism

As shown in Table 1, we can categorize different kinds of parallelism depending on how the data and function are treated.

- If the function is replicated across workers and data is partitioned, we term it data parallelism.
- When the function is partitioned and data is shared or replicated, it's referred to as task parallelism. This involves partitioning the graph into multiple nodes, with each node accessing the same data. In machine learning, this concept is termed model parallelism, with the model representing the function.
- If both function and data are partitioned, it's termed hybrid parallelism.
- Replicating both functions and data is a rare case.

Different domains use varying terminologies for similar concepts. In the architecture and parallel computing domain, operations typically occur within a single node. Here, the focus is on utilizing threads and cores to parallelize the workload effectively. On the other hand, in distributed systems, the coordination extends beyond a single node. Here, multiple computers are synchronized and coordinated, often across data centers or geographically dispersed regions.

In parallel computing, terms such as SIMD (Single Instruction, Multiple Data), MIMD (Multiple Instruction, Multiple Data), and SIMT (Single Instruction, Multiple Threads) are commonly used. These terms denote different levels of parallelism, where "S" stands for single, "M" for multiple, "I" for instruction, and "D" for data. Operations are executed at the instruction level within processors.

For instance, SIMD, akin to data parallelism in machine learning, involves executing a single instruction across multiple data points simultaneously. Moving up a layer to SIMT, the addition of "T" (threads) generalizes SIMD by accommodating multiple threads. At the distributed level, the concept transforms into SPMD (Single Program, Multiple Data), where programs are deployed across processes or nodes. Here, the focus shifts from single instructions to entire programs, reflecting a broader scope of parallel execution.

In the machine learning community, terminology shifts towards data and model parallelism, which are sometimes called inter-operator and intra-operator parallelism, respectively. Here, the primary concern is whether the data or function is partitioned, rather than the specifics of instruction-level parallelism.

These terms reflect similar concepts operating at different granularities, from individual instructions within processors to entire programs across distributed systems.

2.2.1 Task Parallelism

Task parallelism is a fundamental form of parallelism that involves splitting up the workload into tasks. In the context of a dataflow graph, tasks represent the units of work to be performed.

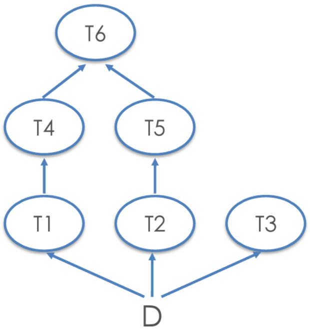


Figure 9: Caption

Consider Figure 9 as an example, where T denotes tasks and D represents data. The graph comprises three primitives $T1$, $T2$, and $T3$ processing the data. Dependencies exist for $T1$ and $T2$, leading to subsequent tasks $T4$ and $T5$ respectively, with the final result merged at $T6$.

Suppose we have 3 workers, where workers are abstract concepts that can represent threads, cores, or nodes.

- Initially, the entire data D is replicated or shared among all workers. This distribution depends on implementation details; for instance, if data is distributed across regions, replication is necessary. Workers within the same data center may access shared data via a common file system.
- Tasks $T1$, $T2$ and $T3$ are assigned to workers $W1$, $W2$ and $W3$ respectively, and executed in parallel.
- Upon completion of $T1$ and $T2$, tasks $T4$ and $T5$ proceed on $W1$ and $W2$ respectively. $W3$ becomes idle after $T3$ finishes.
- Finally, $T6$ is executed on $W1$ to merge the partial results, necessitating communication to synchronize the outcomes from different workers.

Task parallelism exhibits various patterns depending on the graph's partitioning. Typically, a topological sort of tasks is performed for scheduling, followed by the introduction of workers.

Task parallelism offers simplicity, as it involves assigning different workloads to different workers. However, implementing it on complex graphs can be challenging due to the need for synchronization at various points. Additionally, idle time may occur if the dataflow graph is imbalanced, resulting in uneven workload distribution, known as the "bubble."

The degree of parallelism is crucial for determining the number of workers needed. It measures the maximum concurrency achievable in the task graph, indicating how many tasks can run simultaneously. In the provided example shown in Figure 9, the degree of parallelism is 3. Using more than 3 workers wouldn't yield speedup following this partitioning. Additionally, the degree of parallelism decreases over time in this example. Cases vary depending on partition patterns.

To quantify the runtime performance benefits of task parallelism, we calculate the speedup factor.

$$speedup = \frac{Completion\ time\ given\ only\ 1\ worker}{Completion\ time\ given\ n\ (>1)\ workers}$$

Ideally, achieving a speedup of n with n workers is desirable. However, actual performance heavily depends on factors such as the degree of parallelism, task dependency graph structure, intermediate data sizes, and

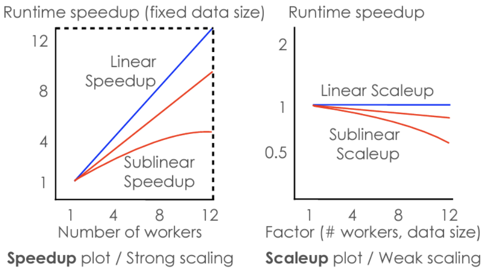


Figure 10: Strong and Weak Scaling

more. In the previous example depicted in Figure 9, it's evident that achieving a speedup of 3 with 3 workers is unlikely due to factors like bubbles and communication overhead.

Even with a balanced dataflow graph featuring n branches with equal workloads, a speedup of n with n workers is not guaranteed. Overheads for synchronization may still be required. Coordination extends beyond communication between branches of different workers; considerations may also include scenarios like dealing with malicious workers and other complexities.

When discussing scalability, two common types of scaling are often considered.

Strong scaling measures how much runtime speedup is achieved with an increasing number of workers while maintaining a fixed total problem size. Ideally, linear speedup is achieved, with the speedup being equal to the number of workers. However, achieving this ideal scenario is challenging in practice due to factors such as heterogeneous graph structures and communication overhead, as depicted in Figure 10.

Weak scaling on the other hand, measures the runtime speedup achieved with an increasing number of workers while maintaining a fixed problem size per worker. In an ideal scenario, we would observe a horizontal line indicating linear scaleup, as shown in Figure 10, signifying no slowdown on each worker. However, in reality, performance tends to degrade as the number of workers increases due to the amplified communication overhead.

***Follow-up question**: Is **superlinear** speedup/scaleup ever possible?*