CS 4440 A

Emerging Database Technologies

Lecture 12

02/19/24

Recap: Approaches to Concurrency Control

Lock-based CC

- 2PL
- Multiple granularity

Optimistic CC

- CC by validation
- Time-stamp-based CC
  - Not covered, Chapter 18.8

Recap: ACID properties

- **Atomicity**: A transaction is an atomic unit of processing; it is either performed in its entirety or not performed at all.
- **Consistency**: A correct execution of the transaction must take the database from one consistent state to another.
- **Isolation**: A transaction should not make its updates visible to other transactions until it is committed.
- **Durability**: Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failure.

Ensuring atomicity and durability with logging and recovery manager

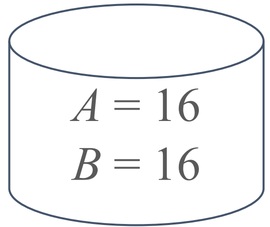
Recovery using undo logging

- Simplifying assumption: use entire log, no matter how long

	Memory			Disk		
Action	t	A	B	A	B	Log
						<START T>
READ(A, t)	8	8		8	8	
$t := t * 2$	16	8		8	8	
WRITE(A, t)	16	16		8	8	<T, A, 8>
READ(B, t)	8	16	8	8	8	
$t := t * 2$	16	16	8	8	8	
WRITE(B, t)	16	16	16	8	8	<T, B, 8>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	
						<COMMIT T>
FLUSH LOG						

Crash

Recovery



Ignore (T was committed)

Ignore (T was committed)

Observe <COMMIT T> record

Nonquiescent checkpointing

- Motivation: avoid shutting down system while checkpointing
- Checkpoint all active transactions, but allow new transactions to enter system

<START T1>
<T1, *A*, 5>
<START T2>
<T2, *B*, 10>
<START CKPT (T1, T2)>
<T2, *C*, 15>
<START T3>
<T1, *D*, 20>
<COMMIT T1>
<T3, E, 25>
<COMMIT T2>
<END CKPT>
<T3, F, 30>

Nonquiescent checkpointing

- Motivation: avoid shutting down system while checkpointing
- Checkpoint all active transactions, but allow new transactions to enter system

<START T1>
<T1, *A*, 5>
<START T2>
<T2, *B*, 10>
<START CKPT (T1, T2)>
<T2, *C*, 15>
<START T3>
<T1, *D*, 20>
<COMMIT T1>
<T3, E, 25>
<COMMIT T2>
<END CKPT>
<T3, F, 30>

Crash

If we first meet <END CKPT>, only need to recover until <START CKPT (T1, T2)>

Nonquiescent checkpointing

- Motivation: avoid shutting down system while checkpointing
- Checkpoint all active transactions, but allow new transactions to enter system

<START T1>
<T1, *A*, 5>
<START T2>
<T2, *B*, 10>
<START CKPT (T1, T2)>
<T2, *C*, 15>
<START T3>
<T1, *D*, 20>
Crash
<COMMIT T1>
<T3, E, 25>
<COMMIT T2>
<END CKPT>
<T3, F, 30>

If we first meet <START CKPT (T1, T2)>, only need to recover until <START T1>

#2 Redo logging

- Motivation: undo logging uses many disk I/O's because it cannot commit a transaction without writing all its changes to disk

- Redo logging lets database changes reside in memory longer

- Redo logging ignores incomplete transactions and repeats committed ones
    - Undo logging cancels incomplete transactions and ignores committed ones

- <*T*, *X*, *v*> now means *T* wrote new value *v* for database element *X*

- One rule: all log records (e.g., <*T*, *X*, *v*> and <COMMIT *T*>) must appear on disk before modifying any database element *X* on disk

Redo logging

- Example

		Memory		Disk		
Action	*t*	*A*	*B*	*A*	*B*	Log
						<START *T*>
READ(*A*, t)	8	8		8	8	
t := *t* * 2	16	8		8	8	
WRITE(*A*, *t*)	16	16		8	8	<*T*, *A*, 16>
READ(*B*, *t*)	8	16	8	8	8	
t := *t* * 2	16	16	8	8	8	
WRITE(*B*, *t*)	16	16	16	8	8	<*T*, *B*, 16>
						<COMMIT *T*>
FLUSH LOG						
OUTPUT(*A*)	16	16	16	16	8	
OUTPUT(*B*)	16	16	16	16	16	

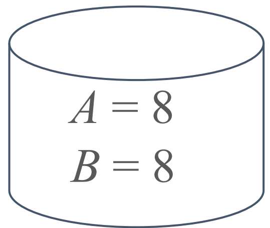
Recovery with redo logging

- Scan log forward and redo committed transactions

Action	Memory t	Memory A	Memory B	Disk A	Disk B	Log
						<START T>
READ(A, t)	8	8		8	8	
$t := t * 2$	16	8		8	8	
WRITE(A, t)	16	16		8	8	<T, A, 16>
READ(B, t)	8	16	8	8	8	
$t := t * 2$	16	16	8	8	8	
WRITE(B, t)	16	16	16	8	8	<T, B, 16>
						<COMMIT T>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	

Crash

Recovery



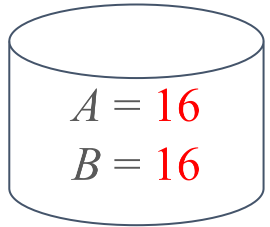
Recovery with redo logging

- Scan log forward and redo committed transactions

	Memory			Disk		
Action	t	A	B	A	B	Log
						<START T>
READ(A, t)	8	8		8	8	
$t := t * 2$	16	8		8	8	
WRITE(A, t)	16	16		8	8	<T, A, 16>
READ(B, t)	8	16	8	8	8	
$t := t * 2$	16	16	8	8	8	
WRITE(B, t)	16	16	16	8	8	<T, B, 16>
						<COMMIT T>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	

Crash

Recovery



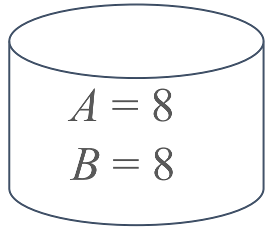
Recovery with redo logging

- Scan log forward and redo committed transactions

Action		Memory		Disk		
Action	t	A	B	A	B	Log
						<START T>
READ(A, t)	8	8		8	8	
$t := t * 2$	16	8		8	8	
WRITE(A, t)	16	16		8	8	<T, A, 16>
READ(B, t)	8	16	8	8	8	
$t := t * 2$	16	16	8	8	8	
WRITE(B, t)	16	16	16	8	8	<T, B, 16>
						<COMMIT T>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	

Crash

Recovery



Recovery with redo logging

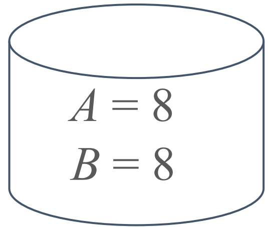
- Scan log forward and redo committed transactions

		Memory		Disk		
Action	t	A	B	A	B	Log
						<START T>
READ(A, t)	8	8		8	8	
$t := t * 2$	16	8		8	8	
WRITE(A, t)	16	16		8	8	<T, A, 16>
READ(B, t)	8	16	8	8	8	
$t := t * 2$	16	16	8	8	8	
WRITE(B, t)	16	16	16	8	8	<T, B, 16>
						<COMMIT T>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	

Crash

Recovery

Do nothing



Nonquiescent checkpointing for redo log

- Write to disk all DB elements modified by committed transactions

<START T1>
<T1, *A*, 5>
<START T2>
<COMMIT T1>
<T2, *B*, 10>
<START CKPT (T2)>

Nonquiescent checkpointing for redo log

- Write to disk all DB elements modified by committed transactions

<START T1>
<T1, *A*, 5>
<START T2>
<COMMIT T1>
<T2, *B*, 10>
<START CKPT (T2)>
<T2, *C*, 15>
<START T3>
<T3, *D*, 20>
<END CKPT>

Write to disk all DB elements by transactions that already committed when START CKPT was written to log (i.e., T1)

Nonquiescent checkpointing for redo log

- Write to disk all DB elements modified by committed transactions

<START T1>
<T1, *A*, 5>
<START T2>
<COMMIT T1>
<T2, *B*, 10>
<START CKPT (T2)>
<T2, *C*, 15>
<START T3>
<T3, *D*, 20>
<END CKPT>
<COMMIT T2>
<COMMIT T3>

Write to disk all DB elements by transactions that already committed when START CKPT was written to log (i.e., T1)

Nonquiescent checkpointing for redo log

- After crash, redo committed transactions that either started after START CKPT or were active during START CKPT

<START T1>
<T1, *A*, 5>
<START T2>
<COMMIT T1>
<T2, *B*, 10>
<START CKPT (T2)>
<T2, *C*, 15>
<START T3>
<T3, *D*, 20>
<END CKPT>
<COMMIT T2>
Crash
<COMMIT T3>

Nonquiescent checkpointing for redo log

- After crash, redo committed transactions that either started after START CKPT or were active during START CKPT

<START T1>
<T1, *A*, 5>
<START T2>
<COMMIT T1>
<T2, *B*, 10>
<START CKPT (T2)>
<T2, *C*, 15>
<START T3>
<T3, *D*, 20>
<END CKPT>
<COMMIT T2>
Crash
<COMMIT T3>

Only redo writes by T2
Write <ABORT T3> in log after recovery

#3 Undo/redo logging

- More flexible than undo or redo logging in ordering actions
- <T, X, v, w> : T changed value of X from v to w
- One rule: <T, X, v, w> must appear on disk before modifying X on disk

Undo/redo logging

- Example

		Memory		Disk		
Action	t	A	B	A	B	Log
						<START T>
READ(A, t)	8	8		8	8	
$t := t * 2$	16	8		8	8	
WRITE(A, t)	16	16		8	8	<T, A, 8, 16>
READ(B, t)	8	16	8	8	8	
$t := t * 2$	16	16	8	8	8	
WRITE(B, t)	16	16	16	8	8	<T, B, 8, 16>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
						<COMMIT T>
OUTPUT(B)	16	16	16	16	16	

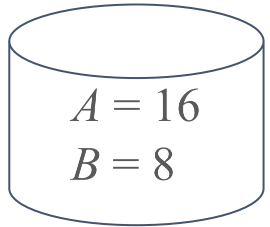
Recovery with undo/redo logging

- Redo all committed transactions and undo all incomplete transactions

Action	t	Memory A	Memory B	Disk A	Disk B	Log
						<START T>
READ(A, t)	8	8		8	8	
$t := t * 2$	16	8		8	8	
WRITE(A, t)	16	16		8	8	<T, A, 8, 16>
READ(B, t)	8	16	8	8	8	
$t := t * 2$	16	16	8	8	8	
WRITE(B, t)	16	16	16	8	8	<T, B, 8, 16>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
						<COMMIT T>
OUTPUT(B)	16	16	16	16	16	

Crash

Recovery



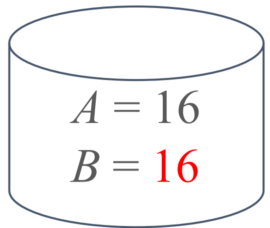
Recovery with undo/redo logging

- Redo all committed transactions and undo all incomplete transactions

Action	Memory			Disk		Log
	t	A	B	A	B	
						<START T>
READ(A, t)	8	8		8	8	
$t := t * 2$	16	8		8	8	
WRITE(A, t)	16	16		8	8	<T, A, 8, 16>
READ(B, t)	8	16	8	8	8	
$t := t * 2$	16	16	8	8	8	
WRITE(B, t)	16	16	16	8	8	<T, B, 8, 16>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
						<COMMIT T>
OUTPUT(B)	16	16	16	16	16	

Crash

Recovery



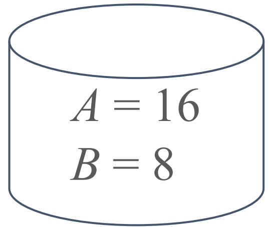
Recovery with undo/redo logging

- Redo all committed transactions and undo all incomplete transactions

	Memory			Disk		
Action	t	A	B	A	B	Log
						<START T>
READ(A, t)	8	8		8	8	
$t := t * 2$	16	8		8	8	
WRITE(A, t)	16	16		8	8	<T, A, 8, 16>
READ(B, t)	8	16	8	8	8	
$t := t * 2$	16	16	8	8	8	
WRITE(B, t)	16	16	16	8	8	<T, B, 8, 16>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
						<COMMIT T>
OUTPUT(B)	16	16	16	16	16	

Crash

Recovery



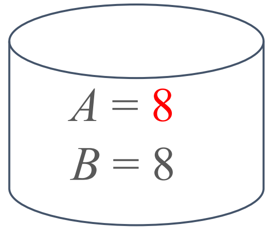
Recovery with undo/redo logging

- Redo all committed transactions and undo all incomplete transactions

		Memory		Disk		
Action	t	A	B	A	B	Log
						<START T>
READ(A, t)	8	8		8	8	
$t := t * 2$	16	8		8	8	
WRITE(A, t)	16	16		8	8	<T, A, 8, 16>
READ(B, t)	8	16	8	8	8	
$t := t * 2$	16	16	8	8	8	
WRITE(B, t)	16	16	16	8	8	<T, B, 8, 16>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
						<COMMIT T>
OUTPUT(B)	16	16	16	16	16	

Crash

Recovery



Nonquiescent checkpointing for undo/redo logging

- Simpler than other logging methods

<START T1>
<T1, *A*, 4, 5>
<START T2>
<COMMIT T1>
<T2, *B*, 9, 10>
<START CKPT (T2)>

Nonquiescent checkpointing for undo/redo logging

- Simpler than other logging methods

<START T1>
<T1, *A*, 4, 5>
<START T2>
<COMMIT T1>
<T2, *B*, 9, 10>
<START CKPT (T2)>
<T2, *C*, 14, 15>
<START T3>
<T3, *D*, 19, 20>
<END CKPT>

Write to disk all the buffers that are dirty

Nonquiescent checkpointing for undo/redo logging

- Simpler than other logging methods

<START T1>
<T1, *A*, 4, 5>
<START T2>
<COMMIT T1>
<T2, *B*, 9, 10>
<START CKPT (T2)>
<T2, *C*, 14, 15>
<START T3>
<T3, *D*, 19, 20>
<END CKPT>
<COMMIT T2>
<COMMIT T3>

Write to disk all the buffers that are dirty

Nonquiescent checkpointing for undo/redo logging

- After a crash, redo committed transactions, and undo uncommitted ones

<START T1>
<T1, *A*, 4, 5>
<START T2>
<COMMIT T1>
<T2, *B*, 9, 10>
<START CKPT (T2)>
<T2, *C*, 14, 15>
<START T3>
<T3, *D*, 19, 20>
<END CKPT>
<COMMIT T2>
Crash
<COMMIT T3>

Nonquiescent checkpointing for undo/redo logging

- After a crash, redo committed transactions, and undo uncommitted ones

<START T1>
<T1, *A*, 4, 5>
<START T2>
<COMMIT T1>
<T2, *B*, 9, 10>
<START CKPT (T2)>
<T2, *C*, 14, 15>
<START T3>
<T3, *D*, 19, 20>
<END CKPT>
<COMMIT T2>
<COMMIT T3>

Crash

Redo T2 by setting C to 15 on disk
(No need to set B to 10 thanks to CKPT)
Undo T3 by setting D to 19 on disk

Summary

Coping with System Failures

- Undo logging
- Redo logging
- Undo/redo logging
- Checkpointing

Today's class

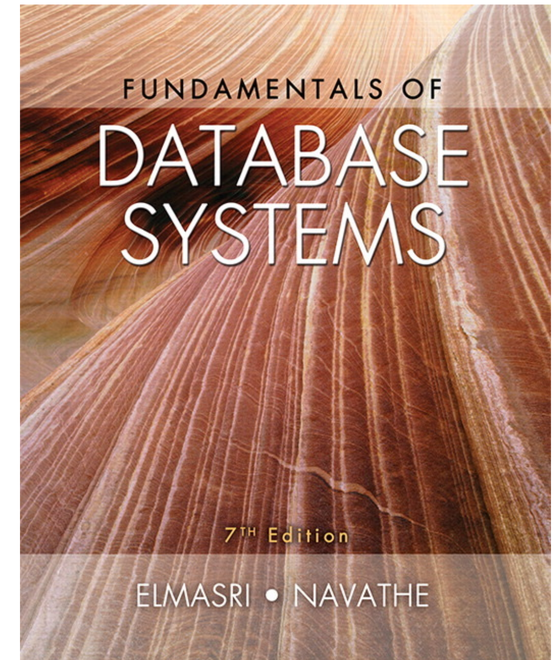
OLAP and Data Warehousing

- OLTP vs OLAP
  - Data Model
  - Storage format
- Data warehouses vs Data Lakes
- Data cubes
- HTAP

Reading Materials

Fundamental of Database Systems (7th Edition)

- Chapter 29 - Overview of Data Warehousing and OLAP



So far we've been dealing with OLTP

OLTP: OnLine Transactional Processing

- Often used to store and manage relevant data to the day-to-day operations of a system or company
  - e.g., ATM transactions, online hotel bookings
- INSERT, UPDATE, DELETE commands
- Handles real-time transactions (response times often in milliseconds)
- ACID properties are often important

This is where relational databases shine!

Ok, so what's OLAP?

OLAP: OnLine Analytical Processing

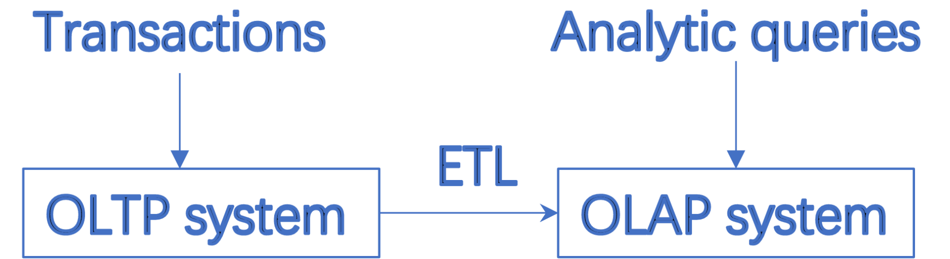
- Also known as decision support or business intelligence (BI), but now BI has grown to include more (e.g., AI)
- A specialization of relational databases that prioritizes the reading and summarizing large volumes (TB, PB) of relational data to understand high-level trends and patterns
  - E.g., the total sales figures of each type of Honda car over time for each county
- "Read-only" queries

Contrast this to OLTP

- "Read-write" queries
- Usually touch a small amount of data
  - e.g., append a new car sale into the sales table

Adapted from Berkeley Info 290T

How is OLAP Performed?



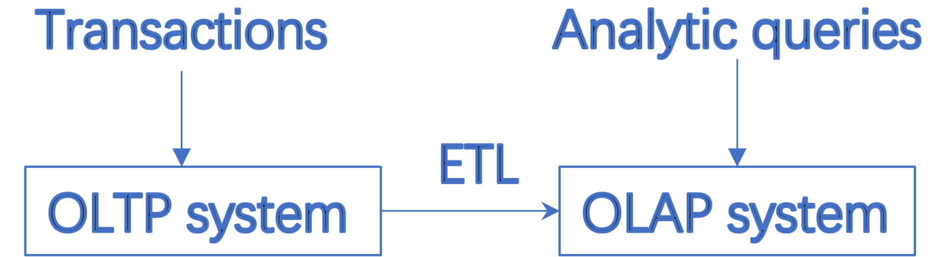
Usually, OLAP is performed on a separate data warehouse away from the critical path of OLTP transactions (a live/transactional database).

This data warehouse is periodically updated with data from various sources (e.g., once an hour or once a day)

- This is through a process of ETL (Extract, Transform, Load)
  - Extract useful business that needs to be summarized, transform it (e.g., canonicalize values, clean it up), load it in the data warehouse
- By doing it periodically, this data warehouse can become stale

Adapted from Berkeley Info 290T

How is OLAP Performed?



Usually, OLAP is performed on a separate data warehouse away from the critical path of OLTP transactions (a live/transactional database).

Why?

- Because OLAP queries end up reading most of the data, and will prevent OLTP queries from taking precedence
  - it is more important to ensure that sales are not prevented than to make sure that a report for a manager is generated promptly
  - the latter will anyway take a long time, so might as well have them wait a bit longer
- It's OK if the warehouse data is a bit stale.

OLAP in Data Warehouses

Here are some data warehouses that you might have heard of

Data Warehouse vs Data Lake

Data warehouse:

- structured data (schema-on-write)
- expensive for large data volumes
- managers and business analysts

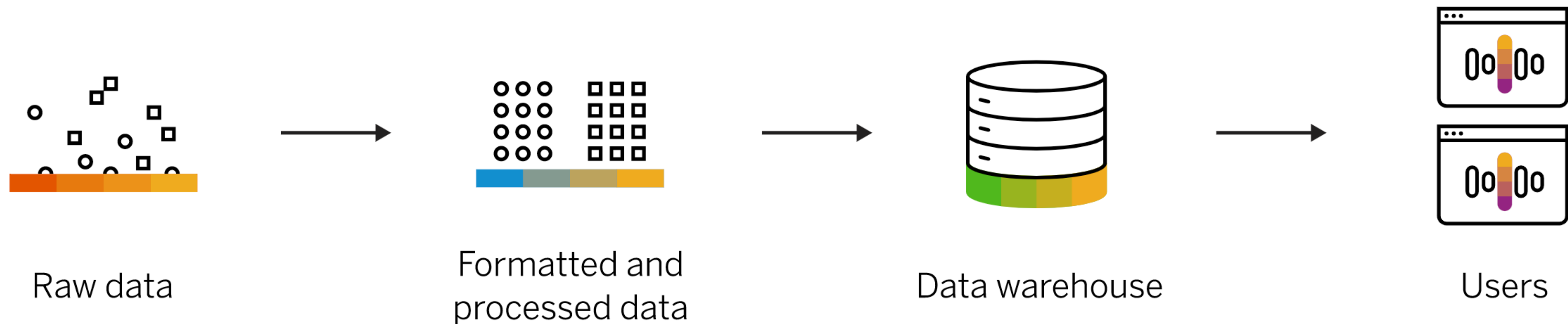


Image source: https://www.sap.com/insights/what-is-a-data-lake.html

Data Lake vs Data Warehouse

Data lake:

- raw data, can be unstructured
- low-cost storage, but no transactions, data quality checks
- data scientists and engineers

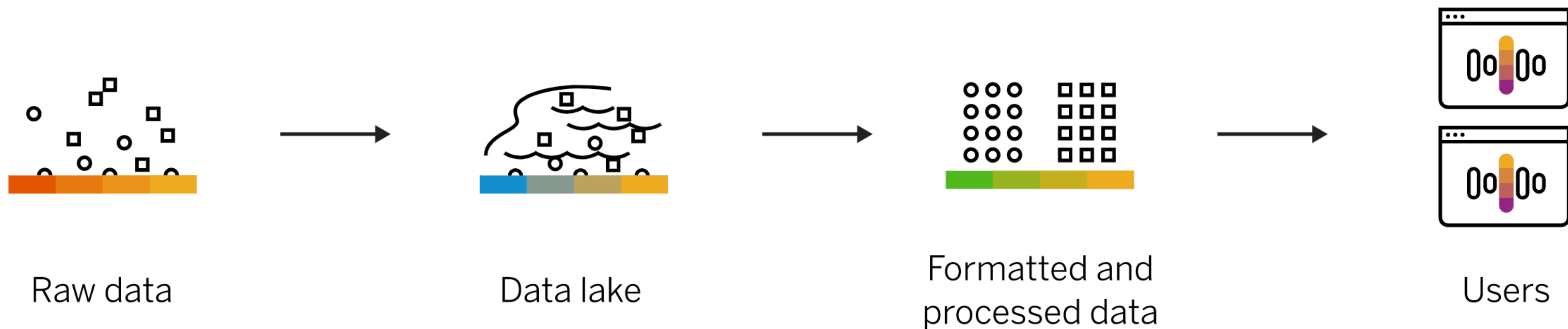


Image source: https://www.sap.com/insights/what-is-a-data-lake.html

We will focus on the following two aspects of OLAP systems

#1 Data Model

- Relational vs multi-dimensional schema

#2 Storage Format

- Row vs column store

#1 Data Model: Multi-dimensional Model

The multi-dimensional data model includes two types of tables:

- Fact table
  - Each tuple is a recorded fact. This fact contains some measured or observed variable (s) and identifies it with pointers to dimension tables. The fact table contains the data, and the dimensions to identify each tuple in the data.
  - A fact table is as an agglomerated view of transaction data whereas each dimension table represents “master data” that those transactions belonged to.
- Dimension table
  - It consists of tuples of attributes of the dimension.

Multi-dimensional Schemas

Star schema:

- Consists of a fact table with a single table for each dimension.

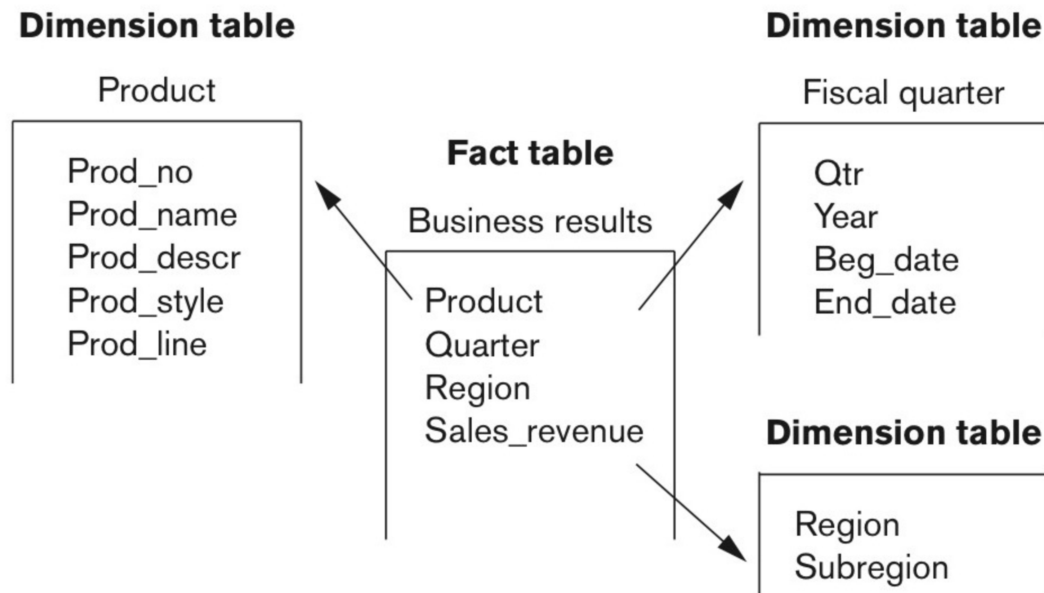


Figure 29.7 A star schema with fact and dimensional tables.

Multi-dimensional Schemas

Snowflake Schema:

- It is a variation of star schema, in which the dimensional tables from a star schema are normalized to eliminate redundancy

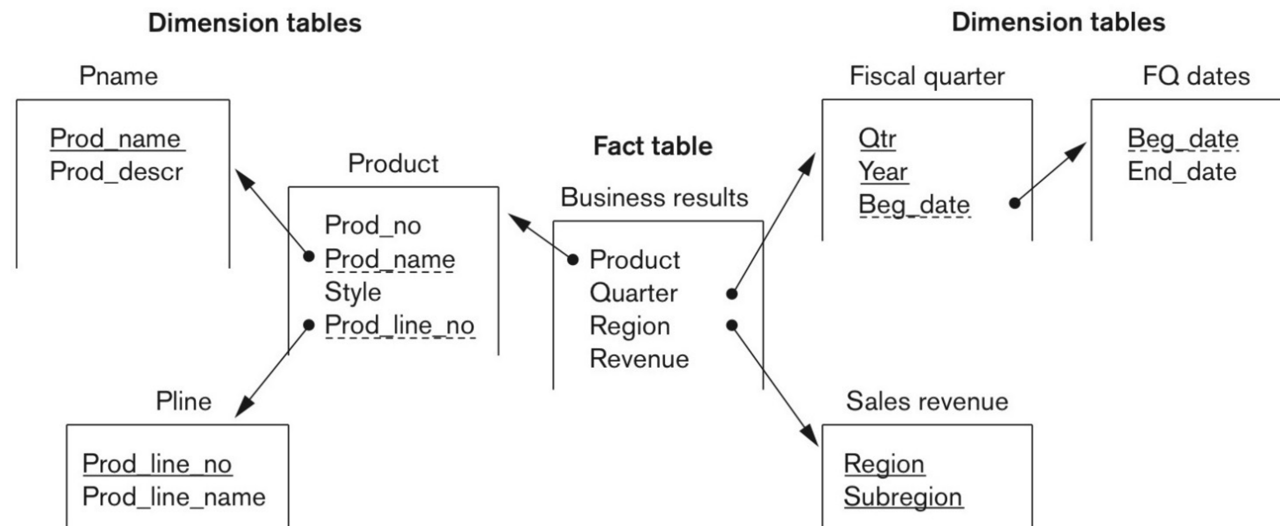


Figure 29.8 A snowflake schema.

Comparison with the Relational Model

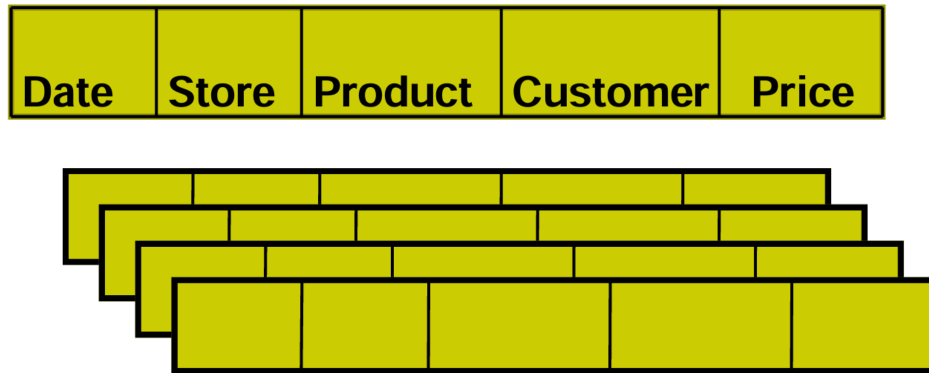
The relational model (used by OLTP) keeps tables normalized

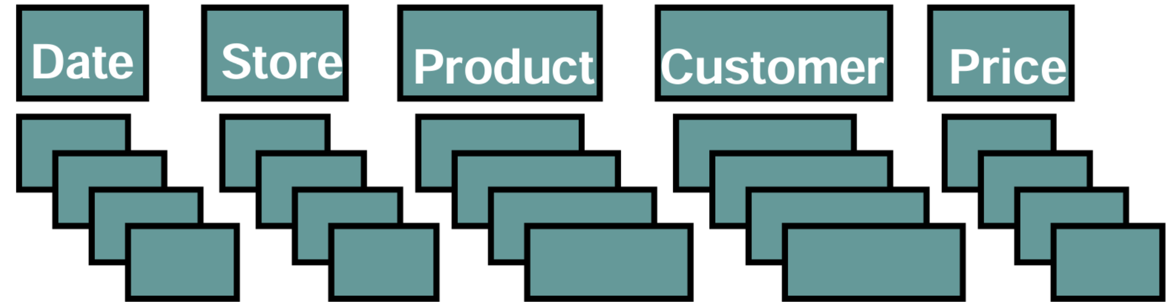
- Minimal updates required for data inserts and deletes => help improve transaction performance

In the multi-dimensional model, the fact table is often in 3NF, but the dimensional tables are denormalized.

- This means columns in a table contains data which is repeated throughout the table => help improve read performance.
- Data is stored in fewer tables, which removes the overhead of having to perform complex joins => helps improve read performance.

#2 Storage Format: Row vs Column Store





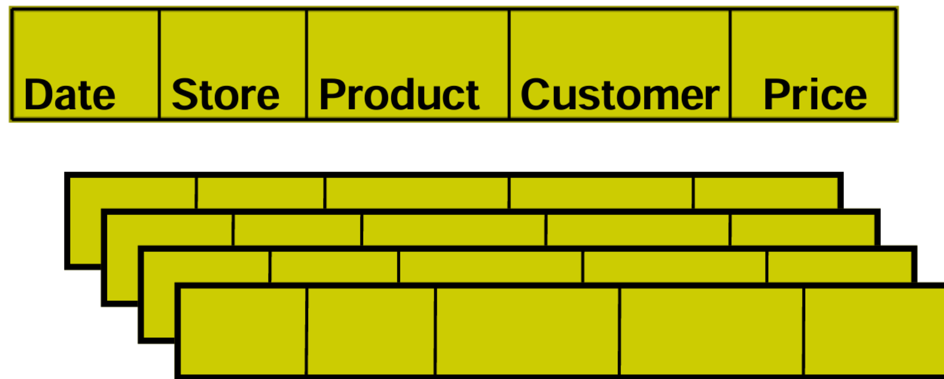
- Store all attributes of a tuple together
- Storage like "row-major order" in a matrix

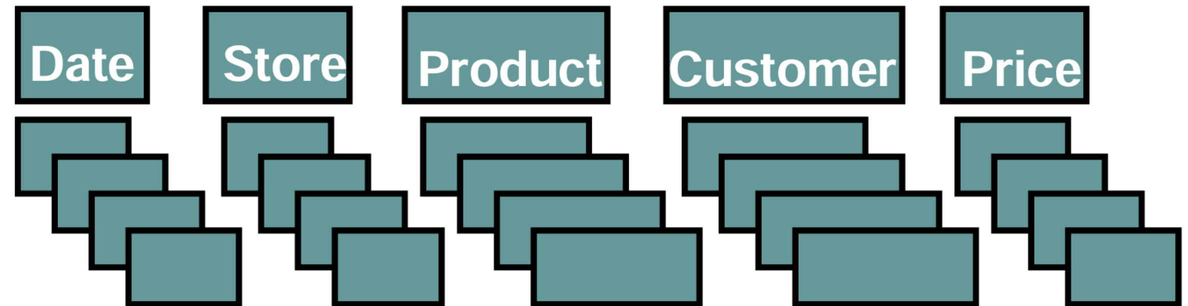
- Store all rows for an attribute together
- Storage like "column-major order" in a matrix

Q: Which format do you think is better suited for OLTP vs OLAP?

Acknowledgement: Slides adapted from VLDB 2009 Tutorial on Column Stores

#2 Storage Format: Row vs Column Store





+ easy to add/modify a record

- need to read unnecessary data

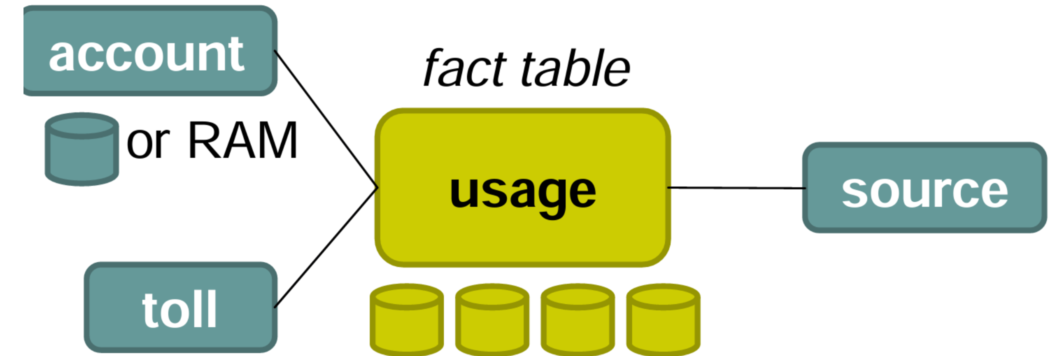
+ only need to read in relevant data

- tuple write require multiple accesses

Acknowledgement: Slides adapted from VLDB 2009 Tutorial on Column Stores

Telco Data Warehousing example

"One Size Fits All? - Part 2: Benchmarking Results"
Stonebraker et al. CIDR 2007



QUERY 2

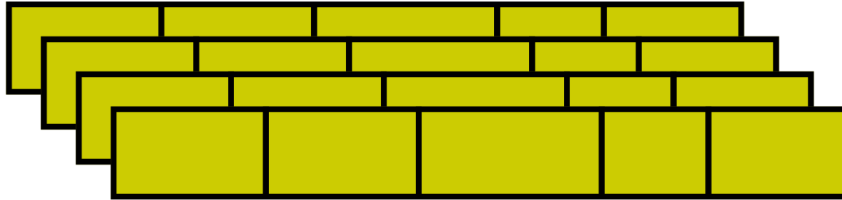
```
SELECT account.account_number,
sum (usage.toll_airtime),
sum (usage.toll_price)
FROM usage, toll, source, account
WHERE usage.toll_id = toll.toll_id
AND usage.source_id = source.source_id
AND usage.account_id = account.account_id
AND toll.type_ind in ('AE'. 'AA')
AND usage.toll_price > 0
AND source.type != 'CIBER'
AND toll.rating_method = 'IS'
AND usage.invoice_date = 20051013
GROUP BY account.account_number
```

	Column-store	*Row-store*
Query 1	*2.06*	*300*
Query 2	*2.20*	*300*
Query 3	*0.09*	*300*
Query 4	*5.24*	*300*
Query 5	*2.88*	*300*

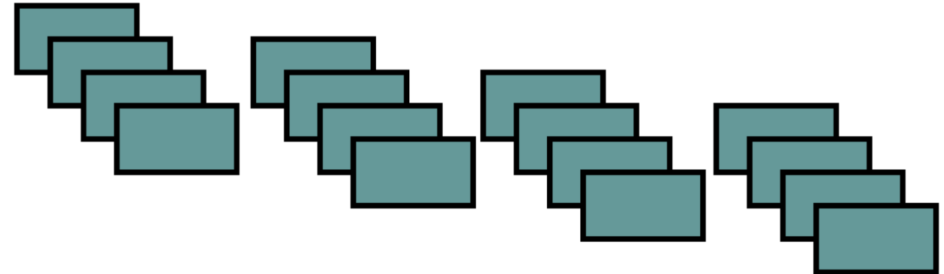
Acknowledgement: Slides adapted from VLDB 2009 Tutorial on Column Stores

Telco example explained: read efficiency

row store



column store



read pages containing entire rows

one row = 212 columns!

is this typical? (it depends)

read only columns needed

in this example: 7 columns

caveats:

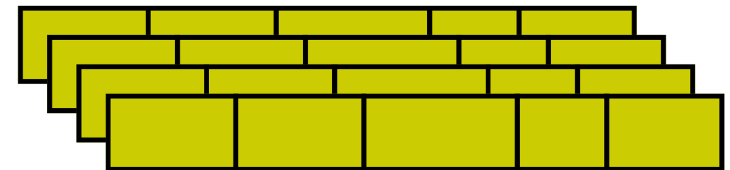
- "select * " not any faster
- clever disk prefetching
- clever tuple reconstruction

Acknowledgement: Slides adapted from VLDB 2009 Tutorial on Column Stores

Telco example explained: compression efficiency

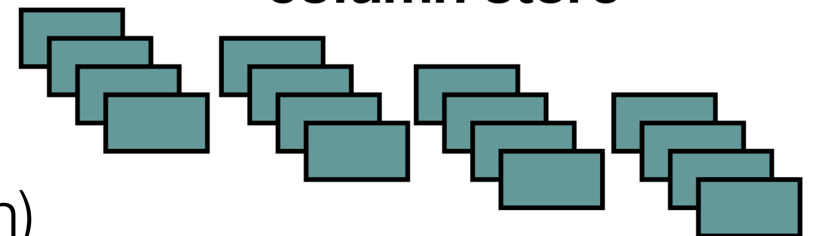
Columns compress better than rows

- Typical row-store compression ratio 1 : 3
- Column-store 1 : 10



Why?

- Rows contain values from different domains
  - More entropy, difficult to dense-pack
- Columns are much more homogeneous
- Caveat: CPU cost (use lightweight compression)



We'll read more about column stores

- C-Store: A Column-oriented DBMS

C-Store: A Column-oriented DBMS

Mike Stonebraker[*], Daniel J. Abadi[*], Adam Batkin[+], Xuedong Chen[†], Mitch Cherniack[+], Miguel Ferreira[*], Edmond Lau[*], Amerson Lin[*], Sam Madden[*], Elizabeth O'Neil[†], Pat O'Neil[†], Alex Rasin[‡], Nga Tran[+], Stan Zdonik[‡]

[*]MIT CSAIL Cambridge, MA

[+]Brandeis University Waltham, MA

[†]UMass Boston Boston, MA

[‡]Brown University Providence, RI

Abstract

This paper presents the design of a read-optimized relational DBMS that contrasts sharply with most current systems, which are write-optimized. Among the many differences in its design are: storage of data by column rather than by row, careful coding and packing of objects into storage including main memory during query processing, storing an overlapping collection of column-oriented projections, rather than the current fare of tables and indexes, a non-traditional implementation of transactions which includes high availability and snapshot isolation for read-only transactions, and the extensive use of bitmap indexes to complement B-tree structures.

We present preliminary performance data on a subset of TPC-H and show that the system we are building, C-Store, is substantially faster than popular commercial products. Hence, the architecture looks very encouraging.

1. Introduction

Most major DBMS vendors implement record-oriented storage systems, where the attributes of a record (or tuple) are placed contiguously in storage. With this *row store* architecture, a single disk write suffices to push all of the fields of a single record out to disk. Hence, high performance writes are achieved, and we call a DBMS with a row store architecture a *write-optimized* system. These are especially effective on OLTP-style applications.

In contrast, systems oriented toward ad-hoc querying of large amounts of data should be *read-optimized*. Data warehouses represent one class of read-optimized system, in which periodically a bulk load of new data is performed, followed by a relatively long period of ad-hoc queries. Other read-mostly applications include customer relationship management (CRM) systems, electronic library card catalogs, and other ad-hoc inquiry systems. In such environments, a *column store* architecture, in which the values for each single column (or attribute) are stored contiguously, should be more efficient. This efficiency has been demonstrated in the warehouse marketplace by products like Sybase IQ [FREN95, SYBA04], Addamark [ADDA04], and KDB [KDB04]. In this paper, we discuss the design of a column store called C-Store that includes a number of novel features relative to existing systems.

With a column store architecture, a DBMS need only read the values of columns required for processing a given query, and can avoid bringing into memory irrelevant attributes. In warehouse environments where typical queries involve aggregates performed over large numbers of data items, a column store has a sizeable performance advantage. However, there are several other major distinctions that can be drawn between an architecture that is read-optimized and one that is write-optimized.

Current relational DBMSs were designed to pad attributes to byte or word boundaries and to store values in their native data format. It was thought that it was too expensive to shift data values onto byte or word boundaries in main memory for processing. However, CPUs are getting faster at a much greater rate than disk bandwidth is increasing. Hence, it makes sense to trade CPU cycles, which are abundant, for disk bandwidth, which is not. This tradeoff appears especially profitable in a read-mostly environment.

There are two ways a column store can use CPU cycles to save disk bandwidth. First, it can code data elements into a more compact form. For example, if one is storing an attribute that is a customer's state of residence, then US states can be coded into six bits, whereas the two-character abbreviation requires 16 bits and a variable length character string for the name of the state requires many more. Second, one should *densepack* values in storage. For example, in a column store it is straightforward to pack N values, each K bits long, into N * K bits. The coding and compressibility advantages of a

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment

Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005

Introducing Data Cubes

Dimensions
Item
season
location

Measure
volume

```
SELECT SUM(volume) …
GROUP BY item, season, location
```

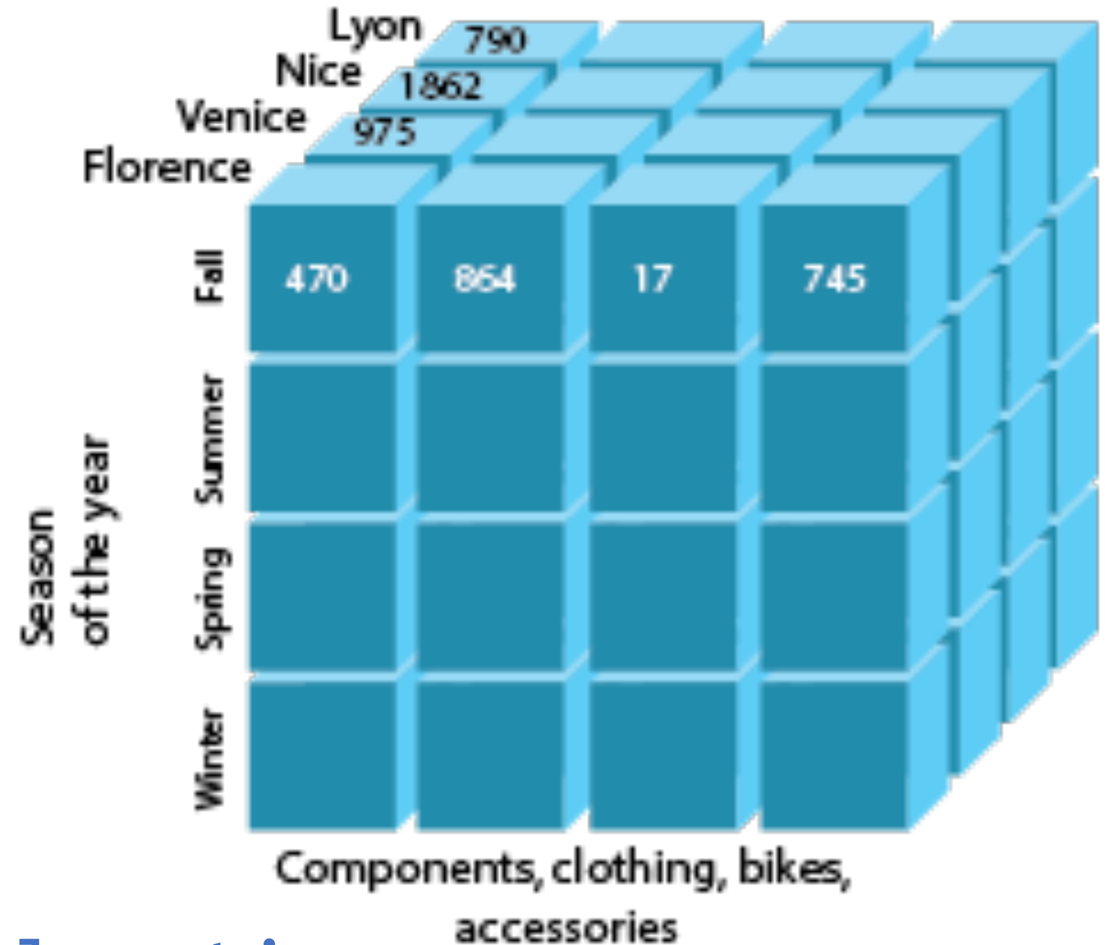
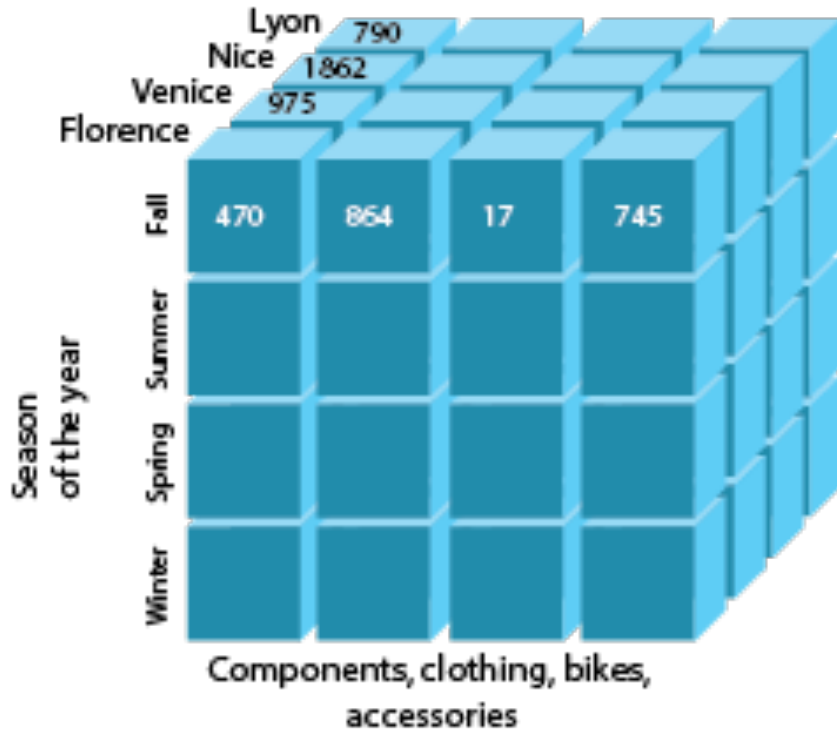


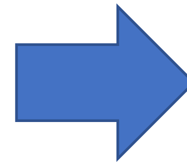
Image source: https://galaktika-soft.com/blog/olap-operations-in-data-mining.html

Operations: Slicing and Dicing

Slicing: select a single value in one of the dimensions to create a smaller cube with one less dimension



Slice for (season = winter)



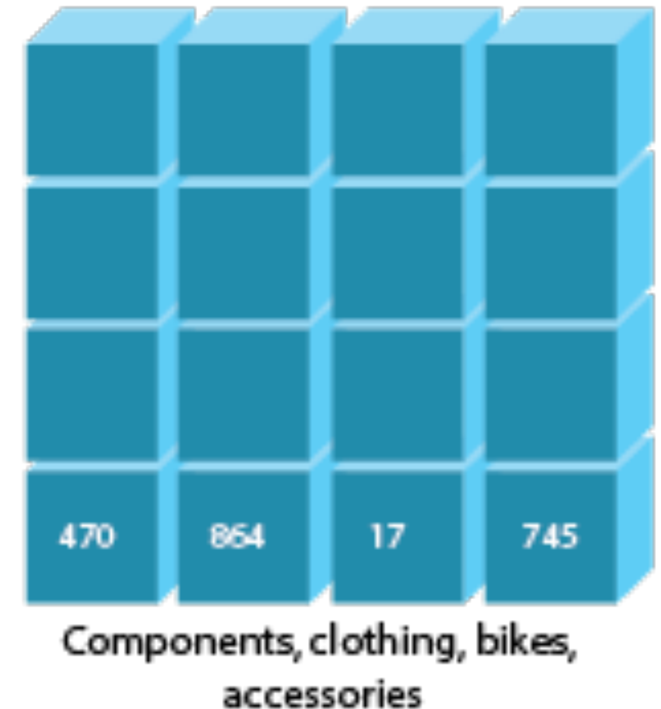
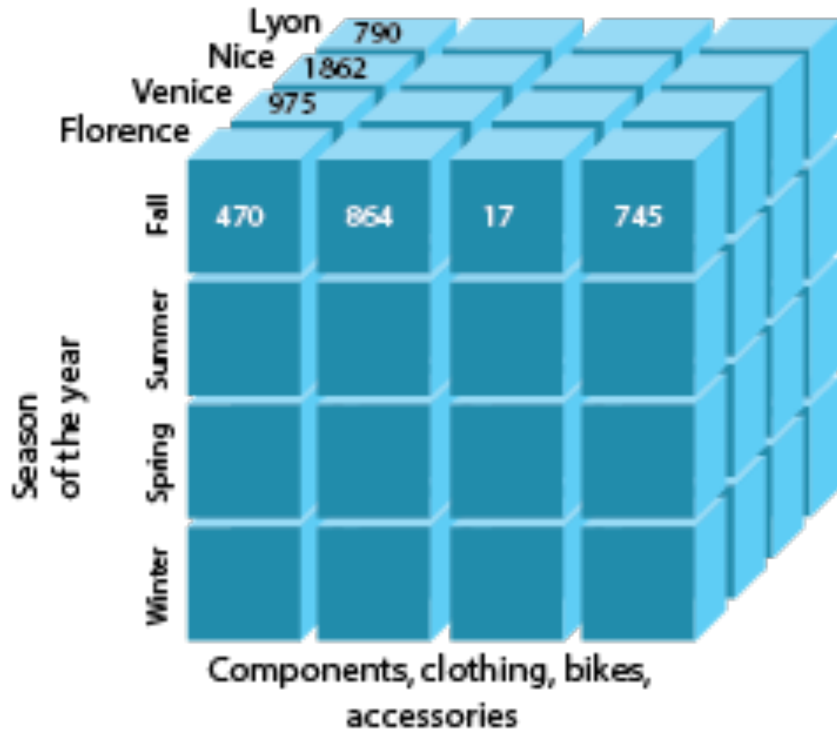


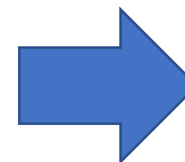
Image source: https://galaktika-soft.com/blog/olap-operations-in-data-mining.html

Operations: Slicing and Dicing

Dicing: select specific values of multiple dimensions to create a new subcube



Dice for
(season = winter or spring)
and
(location = Venice or Florence)
and
(item = components or clothing)



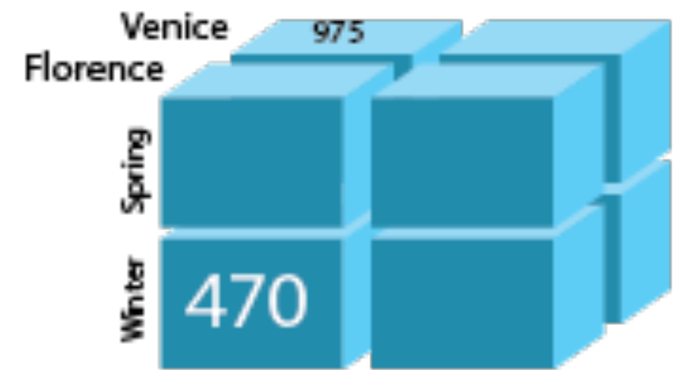
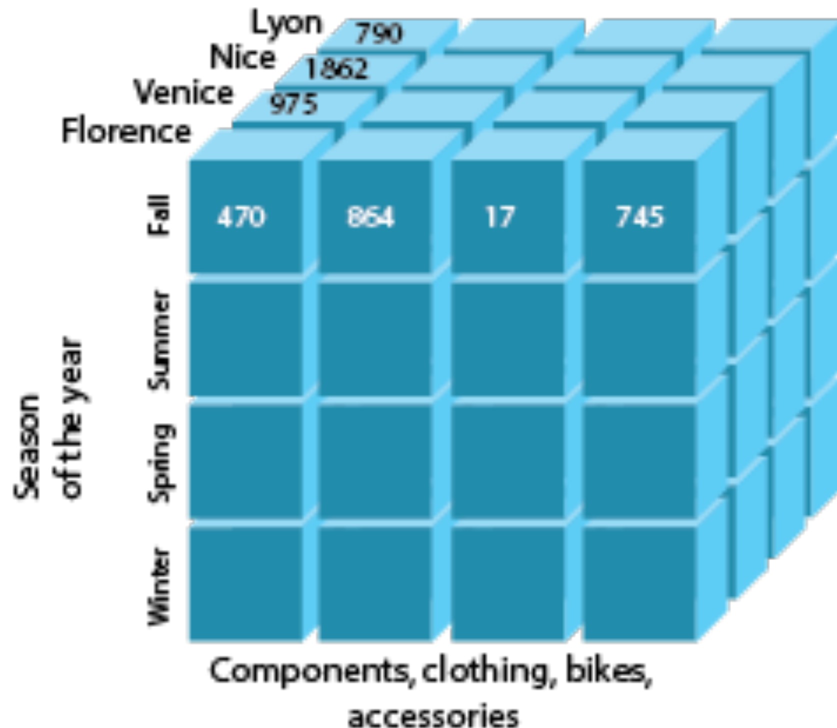


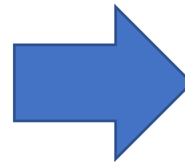
Image source: https://galaktika-soft.com/blog/olap-operations-in-data-mining.html

Operations: Roll up and drill down

Rollup: Moving from a finer to a coarser granularity



Roll up location
(from cities to countries)



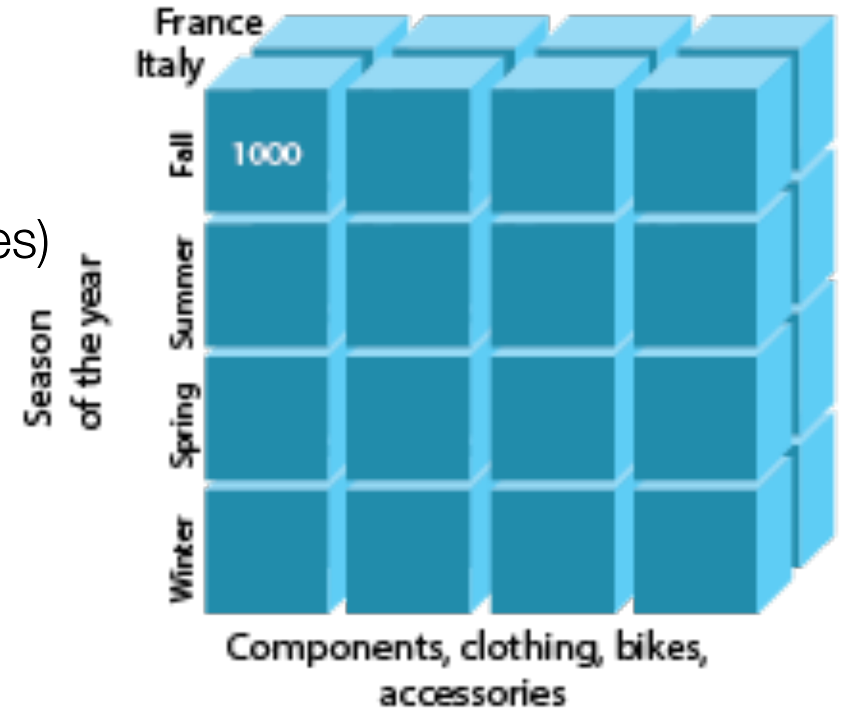
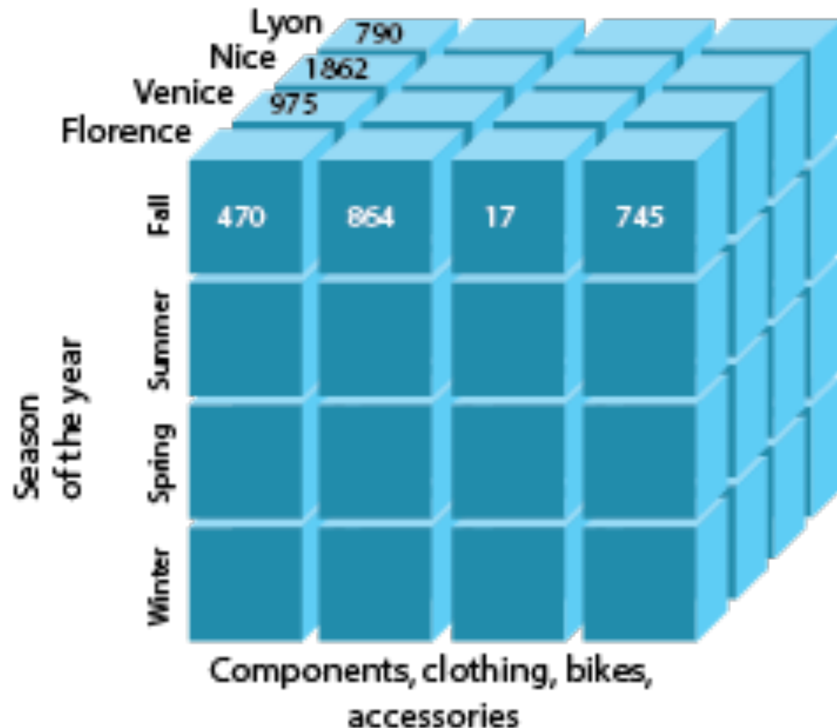


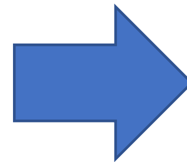
Image source: https://galaktika-soft.com/blog/olap-operations-in-data-mining.html

Operations: Roll up and drill down

Drill-down: Moving from a coarser to a finer granularity



Drill down time
(from quarters to months)



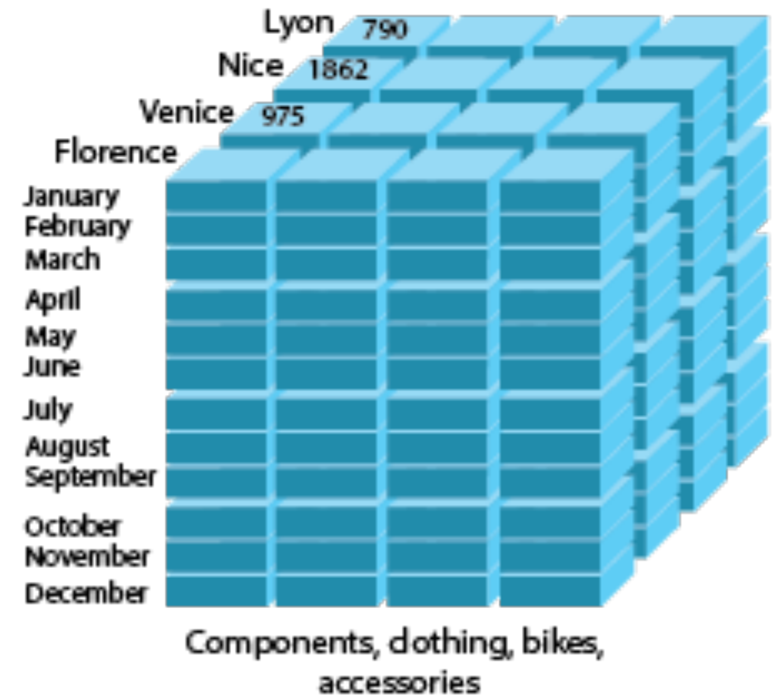


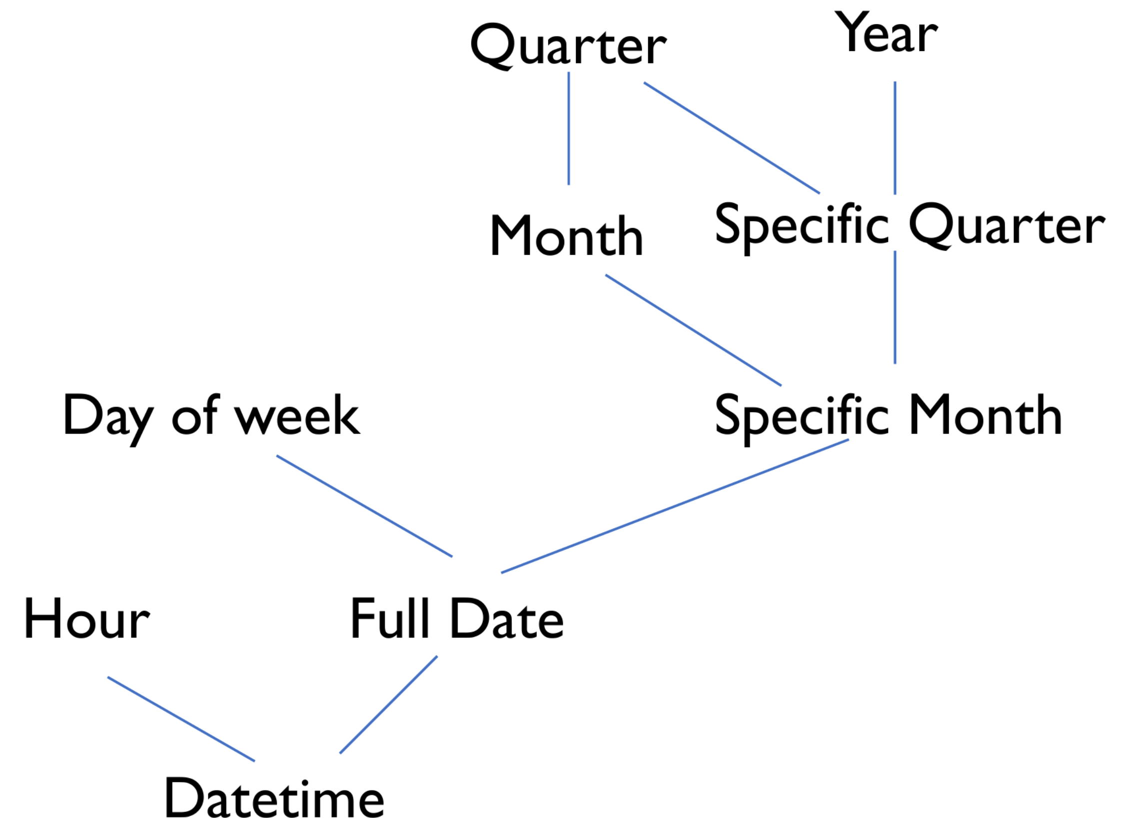
Image source: https://galaktika-soft.com/blog/olap-operations-in-data-mining.html

Partitioning granularity in data cubes

Different partitioning may be useful for different applications

- Q: I want the aggregates per month. What if I set my partitioning based on Full Date and computed the data cube? Can I avoid recomputing the cube?

- Q: What about if I had set my partitioning based on Year?



How to pick the right partitioning granularity?

First, why does this matter?

- If this query is being run once on a petabyte sized warehouse, it is important to get it right!

Think about all the ways you want to slice and dice your data

Pick as fine a granularity of partitioning so that you can recreate all the aggregates, but not so fine as to blow up the query result

- The query result size grows exponentially in the attributes
- This is known as the curse of dimensionality

Adapted from Berkeley Info 290T

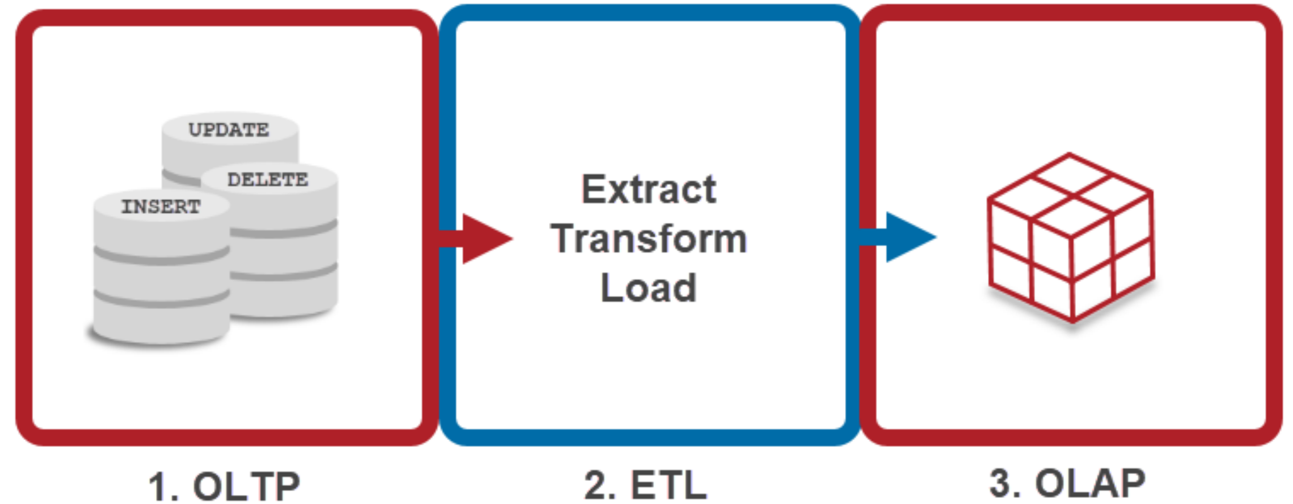
OLAP summary

OLAP is a specialization of relational databases to support analytical processing and report generation

- Typically done in large “batch” operations on the entire database
- Rule of thumb: pick as “coarse-grained” materialized views or query results as will allow you to construct all the cross-tabs that may be necessary

The concepts of OLAP data cubes, hierarchies, slicing/dicing, and rollup/drilldown are valuable to describe what you’re doing when you are exploring your data

OLAP summary



OLAP vs OLTP

- Data Model
- Storage Format

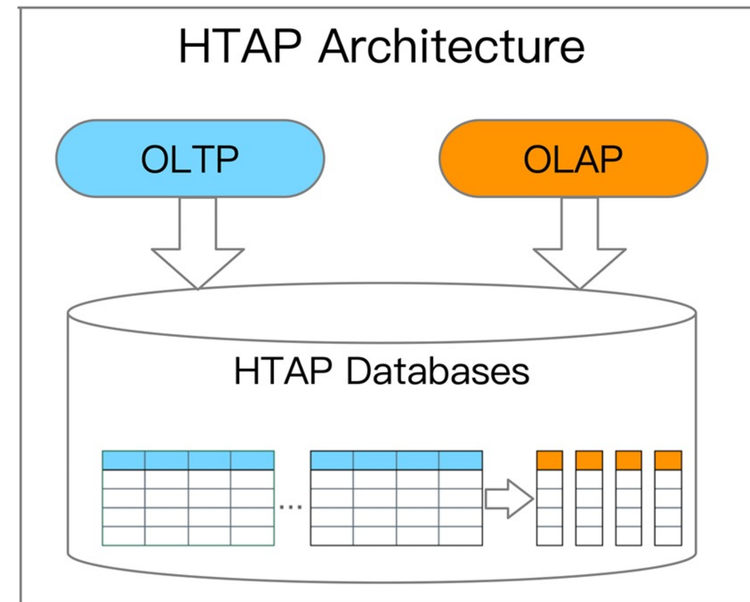
Data warehouses vs Data Lakes

Data cubes

- Slicing and dicing
- Rollup and drill-down
- Selecting partitioning granularity

A brief intro to HTAP Databases

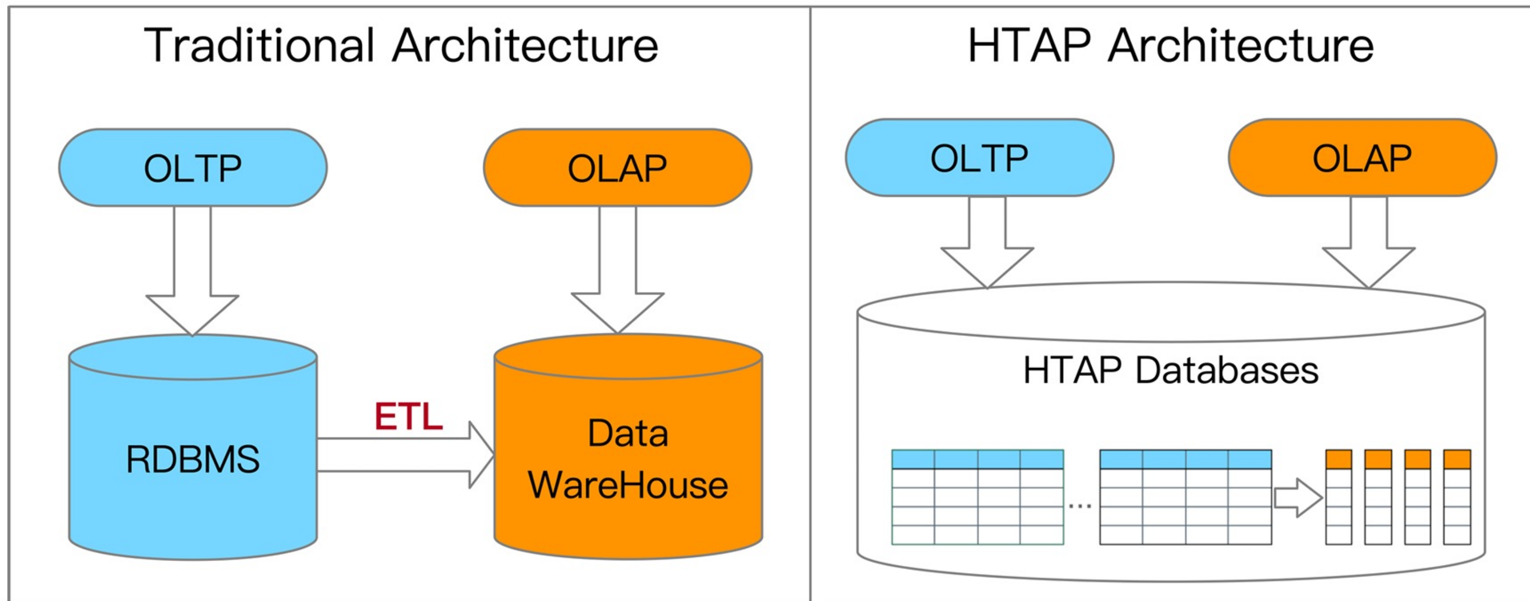
Slides adapted from
SIGMOD'22 Tutorial HTAP Databases: A Tutorial by
Guoliang Li and Chao Zhang



Motivation

HTAP: Hybrid Transaction Analytical Processing

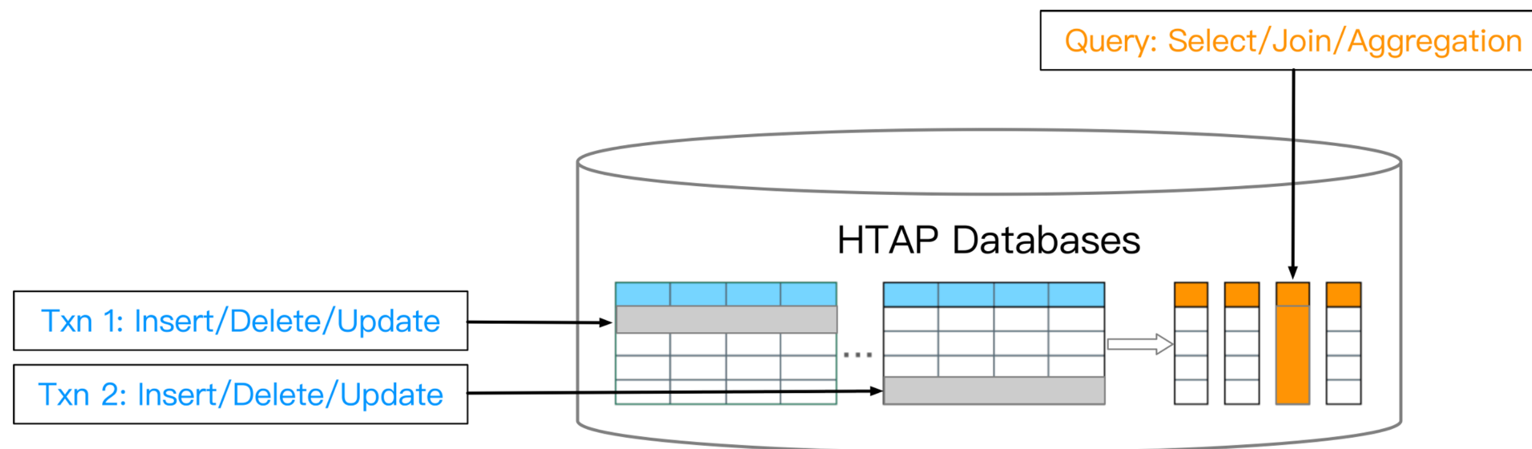
- Gartner's new definition in 2018: supports weaving analytical and transaction processing techniques together as needed to accomplish the business task.



Slides Adapted from SIGMOD'22 Tutorial

Motivation

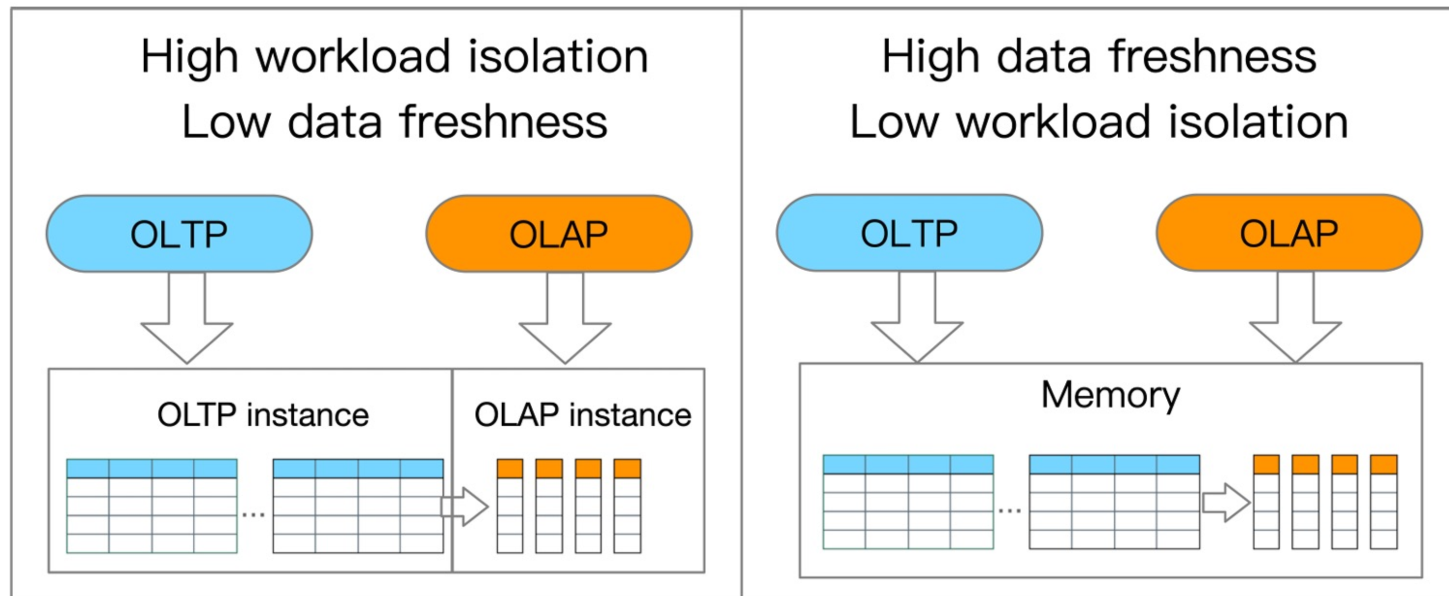
- Rule of thumb 1: row store is ideal for OLTP workloads
  - Row-wise, update-heavy, short-lived transactions
- Rule of thumb 2: column store is best suited for OLAP workload
  - Column-wise, read-heavy, bandwidth-intensive queries



Slides Adapted from SIGMOD'22 Tutorial

A trade-off for HTAP databases

- Workload isolation: the isolation level of handling the mixed workloads
- Data freshness: the portion of latest transaction data that is read by OLAP
- Trade-off for workload isolation and data freshness
  - High workload isolation leads to low data freshness
  - Low workload isolation results in high data freshness



Slides Adapted from SIGMOD'22 Tutorial

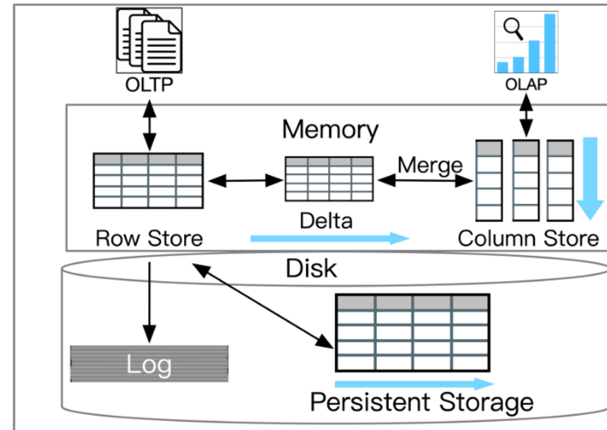
An Overview of HTAP Architectures

(a) **Primary Row Store + In-Memory Column Store**

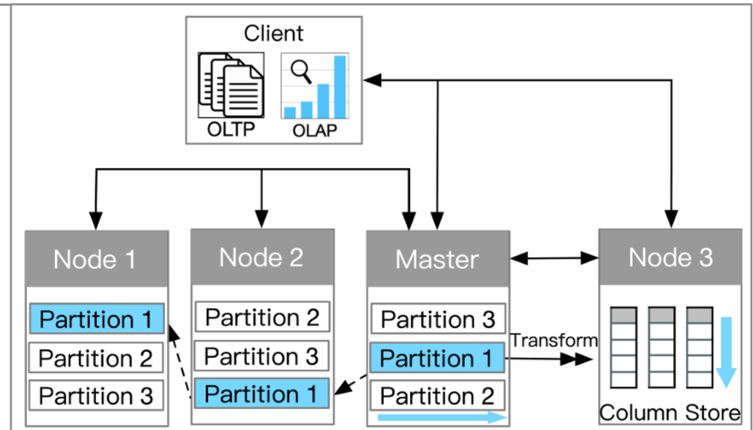
(b) **Distributed Row Store + Column Store Replica**

(c) **Disk Row Store + Distributed Column Store**

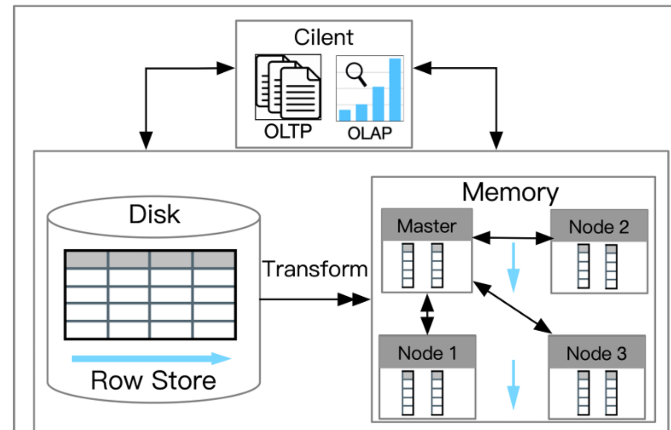
(d) **Primary Column Store + Delta Row Store**



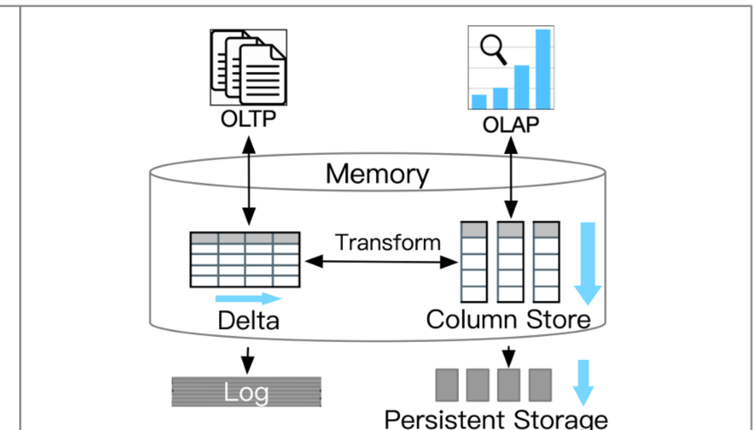
(a) Primary Row Store+In–Memory Column Store



(b) Distributed Row Store+Column Store Replica



(c) Disk Row Store+Distributed Column Store



(d) Primary Column Store+Delta Row Store

Slides Adapted from SIGMOD’22 Tutorial

A summary of HTAP databases

Category	HTAP Databases	OLTP Throughput	OLAP Throughput	OLTP Scalability	OLAP Scalability	Workload Isolation	Data Freshness
Primary Row Store+ In Memory Column Store	Oracle Dual-Format SQL Server, DB2 BLU	High	High	Medium	Low	Low	High
Distributed Row Store + Column Store Replica	TiDB, F1 Lightning SingleStore	Medium	Medium	High	High	High	Low
Disk Row Store + Distributed Column Store	MySQL Heatwave, Oracle RAC	Medium	Medium	Medium	High	High	Medium
Primary Column Store + Delta Row Store	SAP HANA (without scale-out), Hyper	Medium	High	Low	Medium	Low	High

* Readings: HTAP Databases: What is New and What is Next