Indexing & Data Warehouses

CS3200 / CS5200 : Databases

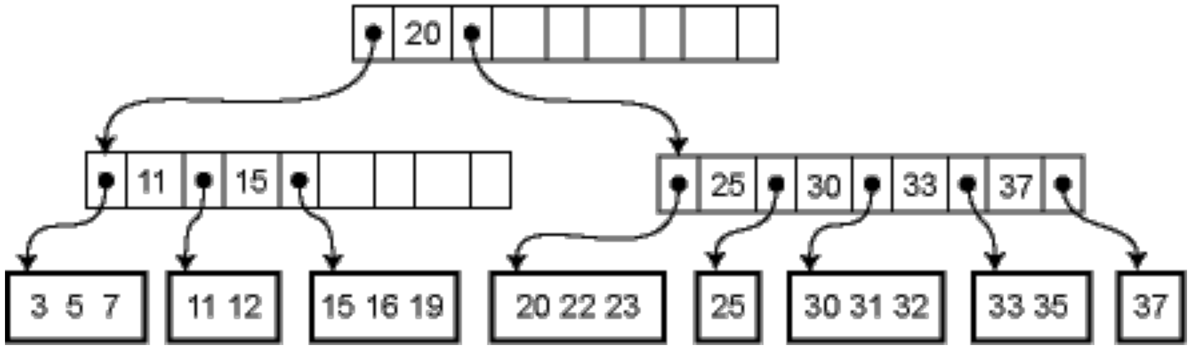
Northeastern University

Indexing

Indexes are a data-structure used to help us locate information in our table more quickly.

Advantage:

Faster read access



Disadvantage:

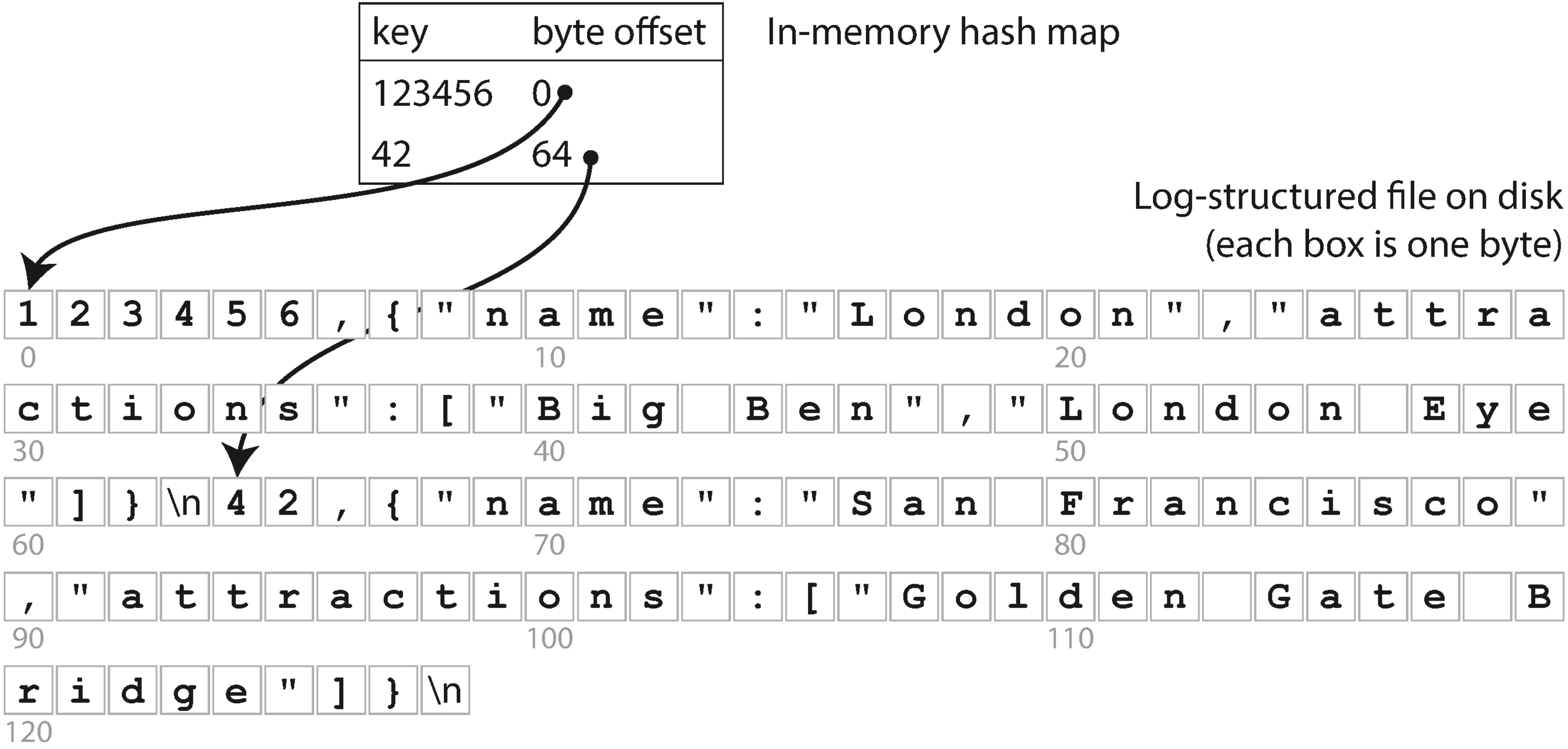
Slower inserts / updates

Storage overhead

More complex backup / recovery requirements

(systems can crash at any moment!)

Hash Indexing (single sequential log file)



Secondary Indexes (Non-unique rows)

Sales transactions

0 → { amount : 20.32, state : VT }
1 → { amount : 34.99, state : MA }
2 → { amount : 12.98, state : VT }
3 → { amount : 93.23, state : VT }
4 → { amount : 50.17, state : CA }

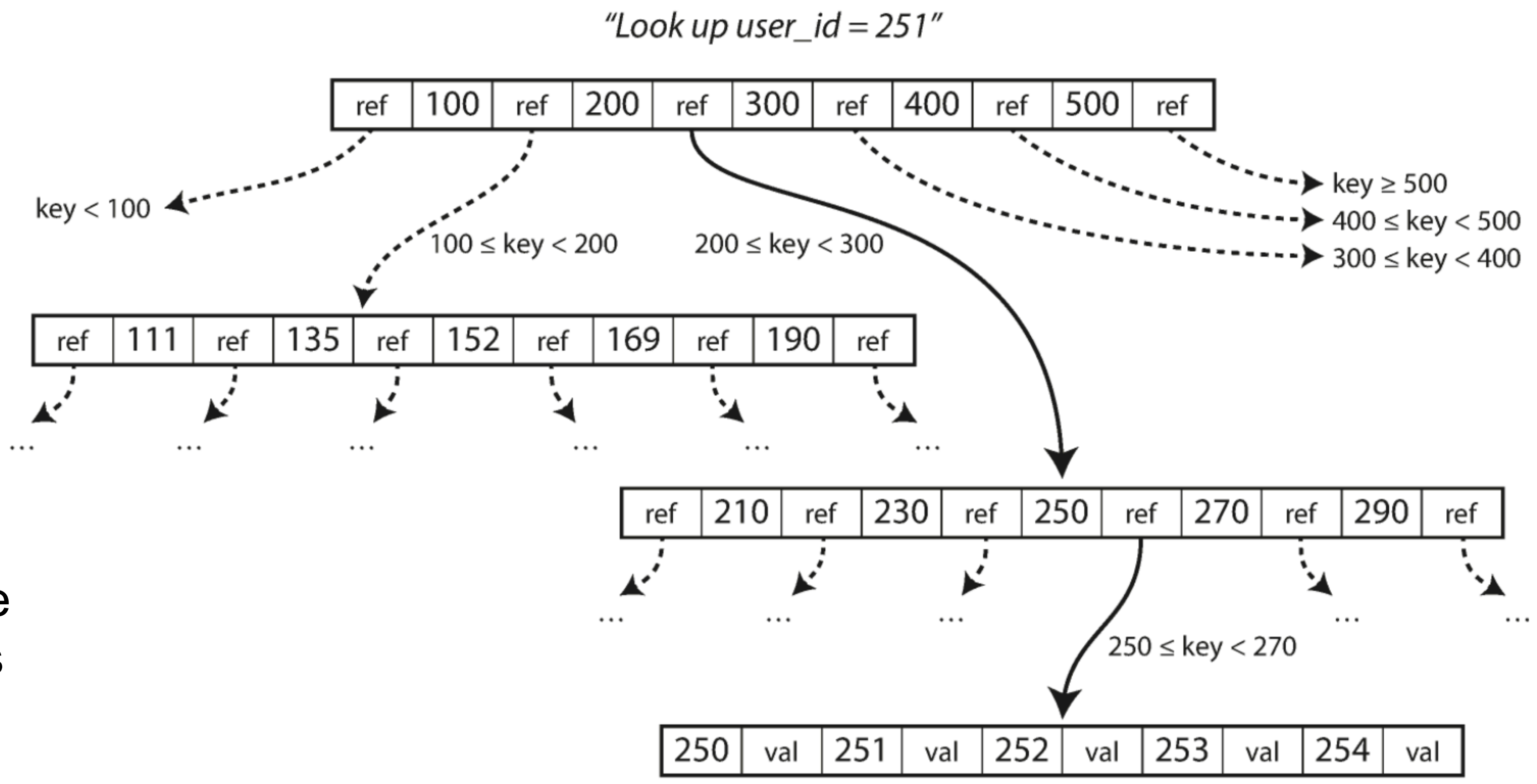
What if we want to find transactions in Vermont?

Secondary Index on State

VT → { transactions : [0, 2, 3] }
MA → { transactions : [1] }
CA → { transactions : [4] }

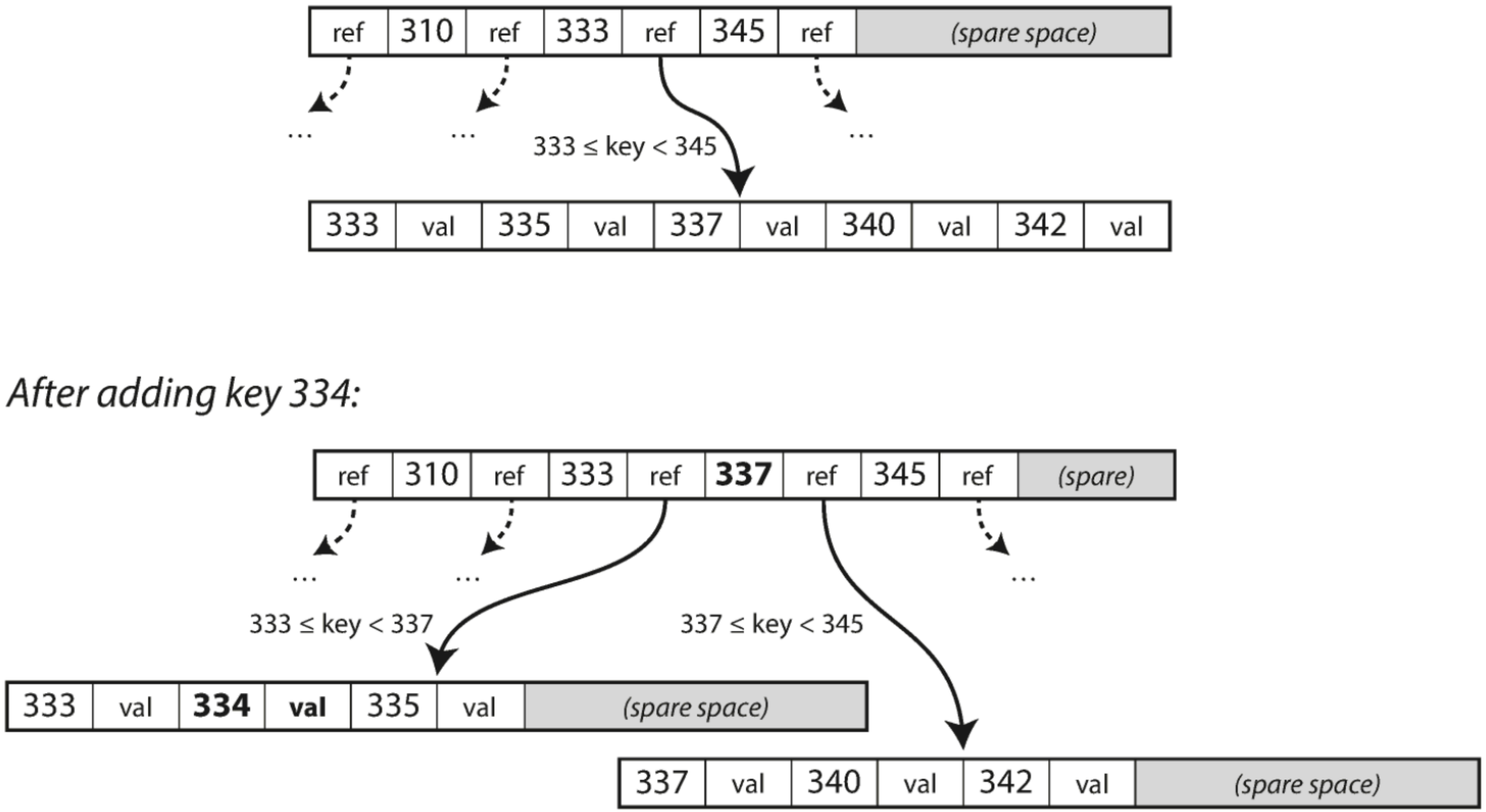
B-Trees

- Most commonly used since their introduction in the 1970s
- Keys are sorted, enabling efficient range queries
- B-Trees organize the DB into fixed sized blocks (pages) typically 4k or more.
- Each child node is responsible for a contiguous range of keys
- Branching factors of several hundred are common



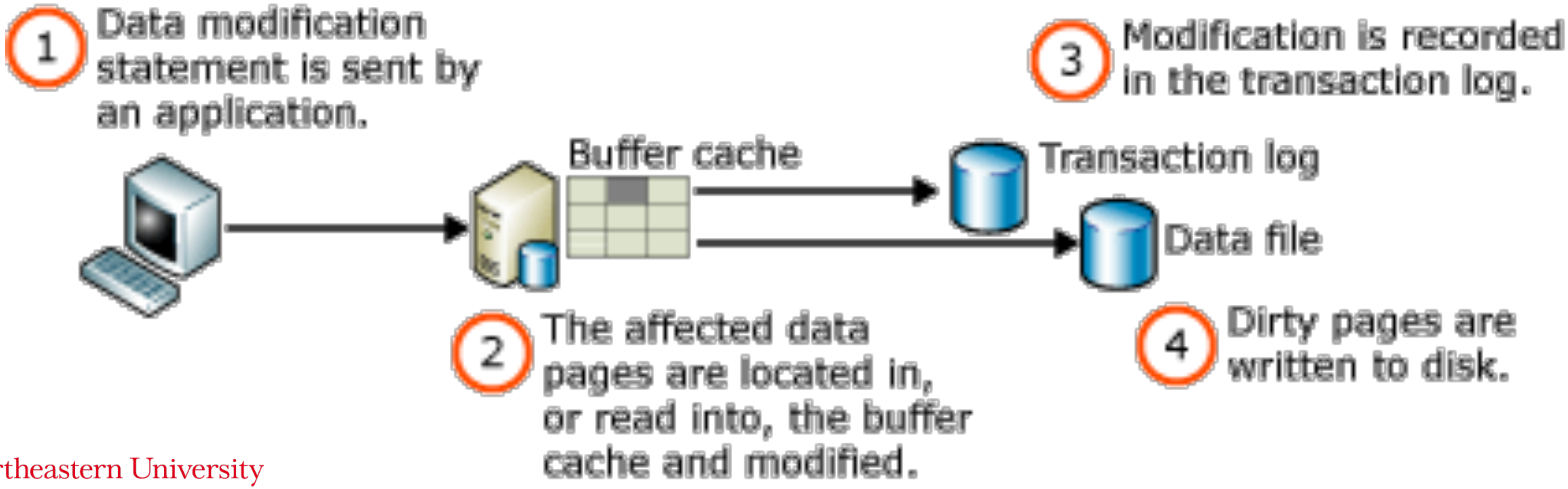
B-Trees: Key addition with page split

- We want to add a value for key 334 but there is no more room in the page
- The page is split and references to the parent page are updated



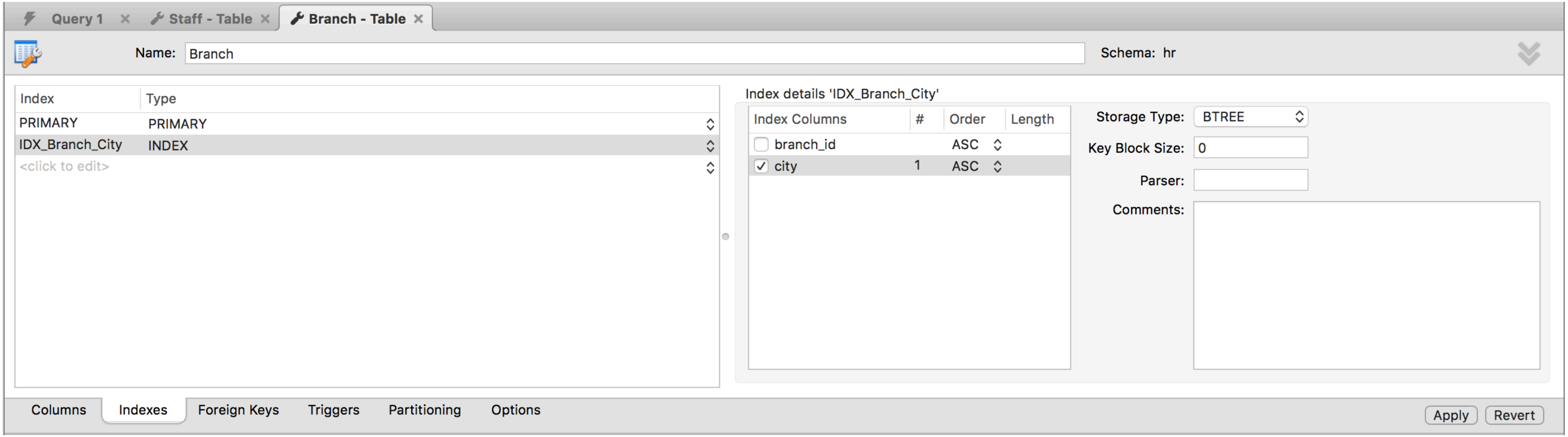
WAL: Write-Ahead Logs

- A system failure could leave the index in a corrupted state. Why?
- Data pages cannot be written to disk until the log records describing those changes are written to disk FIRST.
- Use WAL for crash recovery



Guidelines for when to add indexes

- When a column is used frequently in search conditions or joins
- When the column contains a large number of distinct values
- When the column is updated infrequently
- Table access dominated by SELECT rather than UPDATE or INSERT



Following tweets: An RDB approach (early twitter)

```sql
SELECT tweets.*, users.* FROM tweets
  JOIN users   ON tweets.sender_id    = users.id
  JOIN follows ON follows.followee_id = users.id
  WHERE follows.follower_id = current_user
```

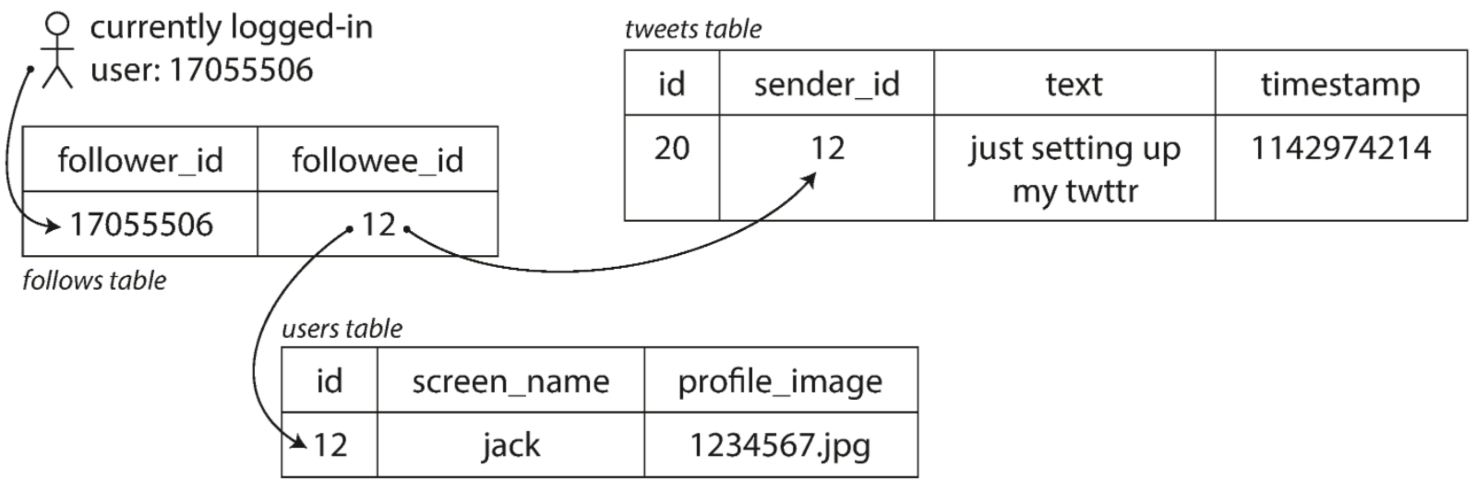


Figure 1-2. Simple relational schema for implementing a Twitter home timeline.

Following tweets: A scalable approach

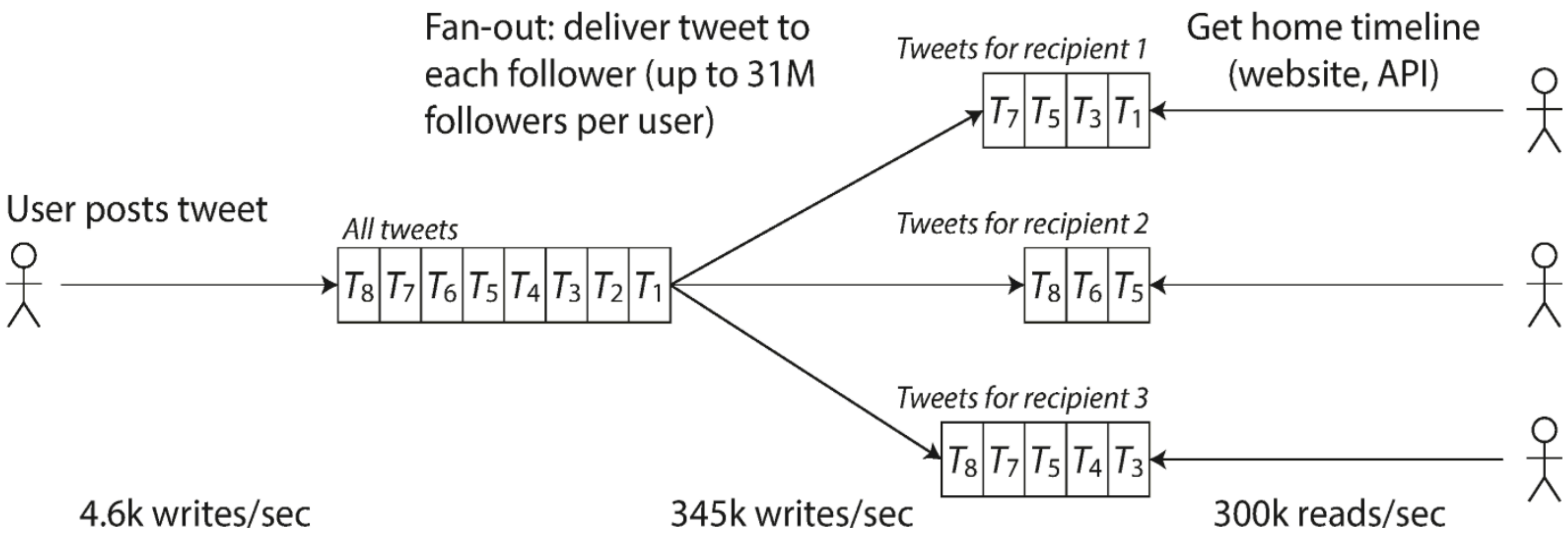


Figure 1-3. Twitter's data pipeline for delivering tweets to followers, with load parameters as of November 2012 [16].

On average a tweet is delivered to 75 followers – and some users have millions of followers!

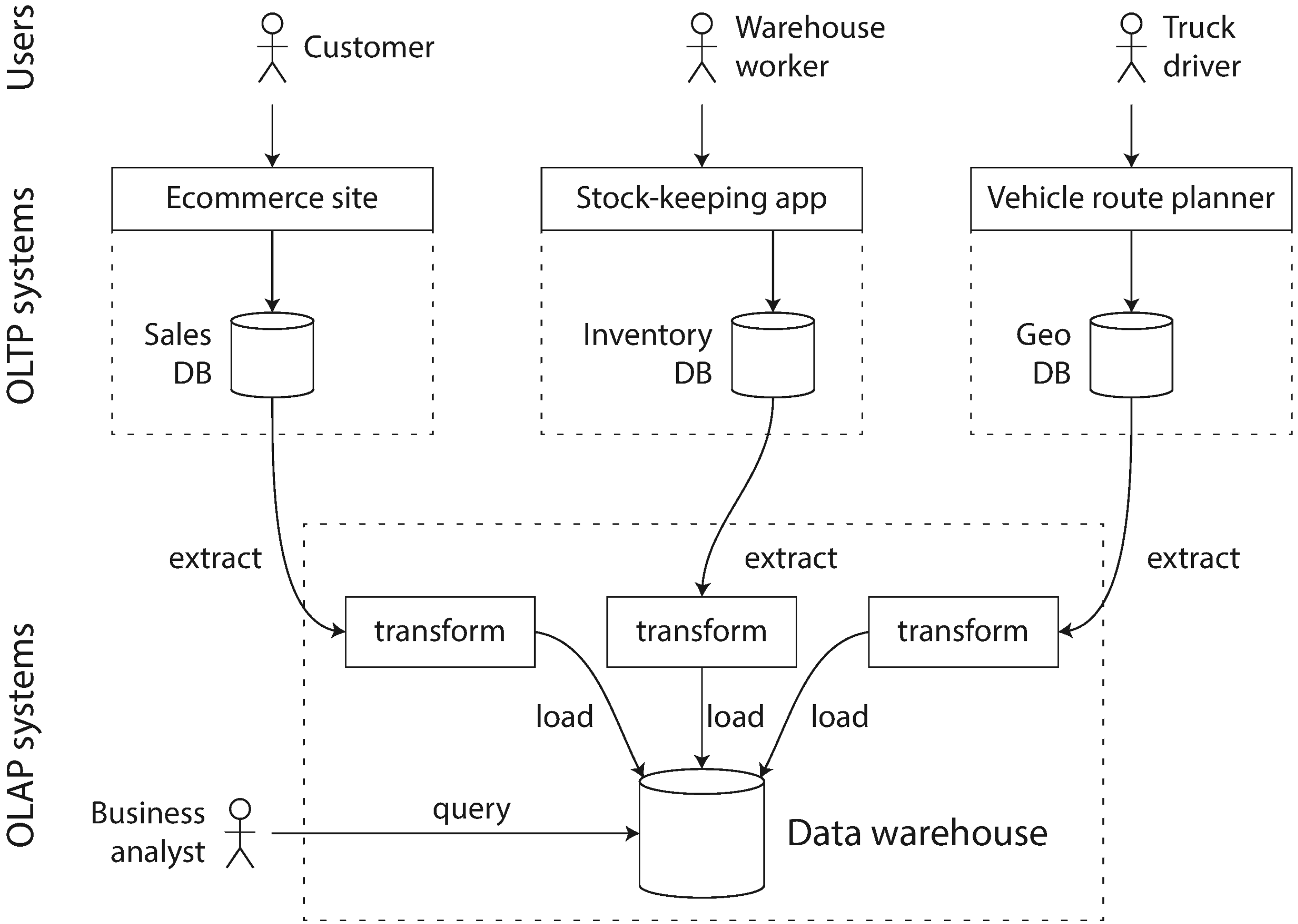
Performance objective: Deliver a tweet within 5 seconds.

Hybrid approach: pre-fetch tweets of celebrities separately and merge them into user's timelines.

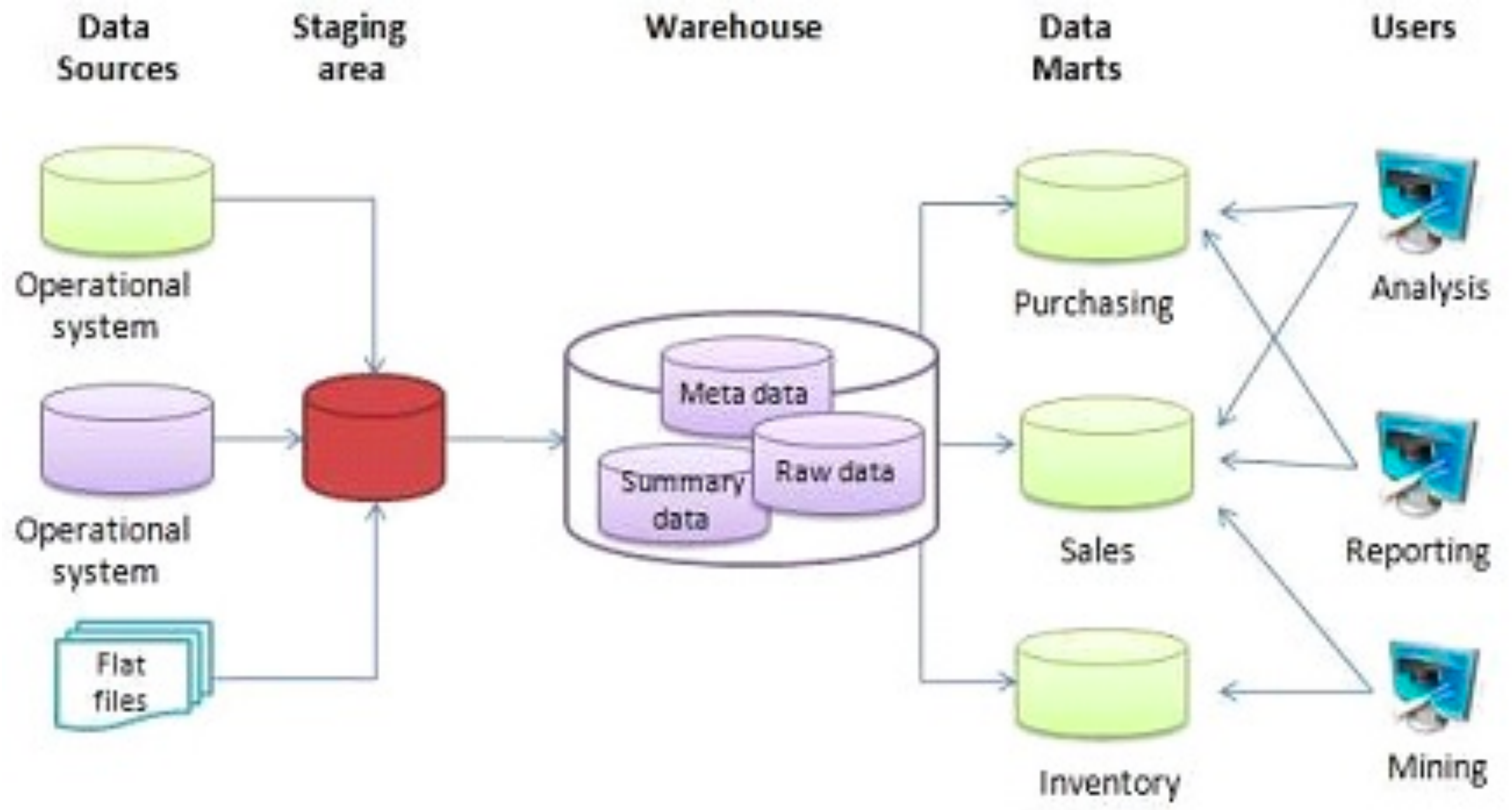
Data Warehousing

A *data warehouse* provides a separate resource for data analytics.

Why is this useful?



Data Warehouse → Data Marts



OLTP vs. OLAP

OLTP

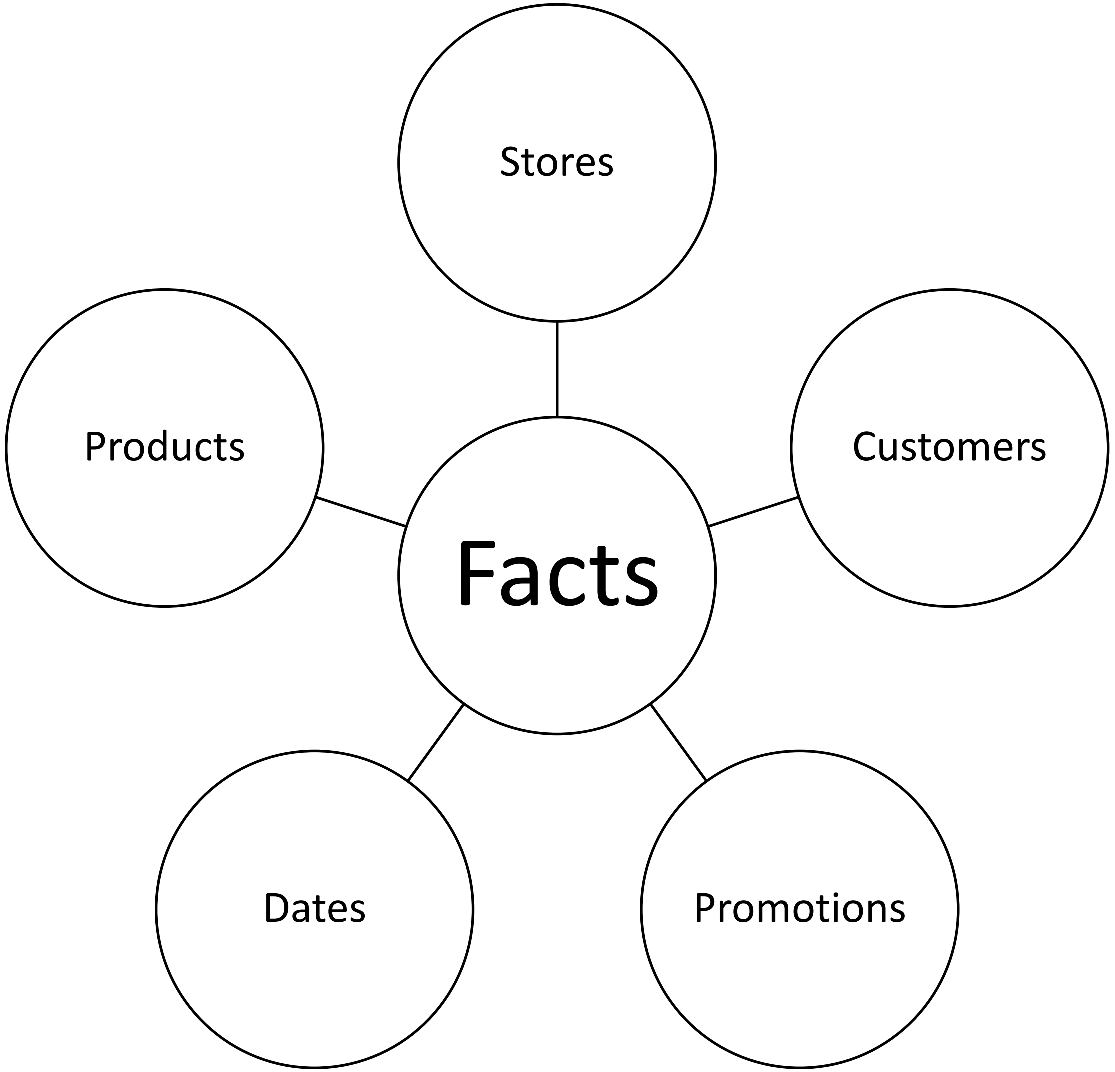
- Interactive
- Low latency
- Row-level operations

OLAP

- Batch-oriented
- Table-level operations
- ETL
- Aggregate Analytics
- Data warehousing

Property	Transaction processing systems (OLTP)	Analytic systems (OLAP)
Main read pattern	Small number of records per query, fetched by key	Aggregate over large number of records
Main write pattern	Random-access, low-latency writes from user input	Bulk import (ETL) or event stream
Primarily used by	End user/customer, via web application	Internal analyst, for decision support
What data represents	Latest state of data (current point in time)	History of events that hap-pened over time
Dataset size	Gigabytes to terabytes	Terabytes to petabytes

Star Schema



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

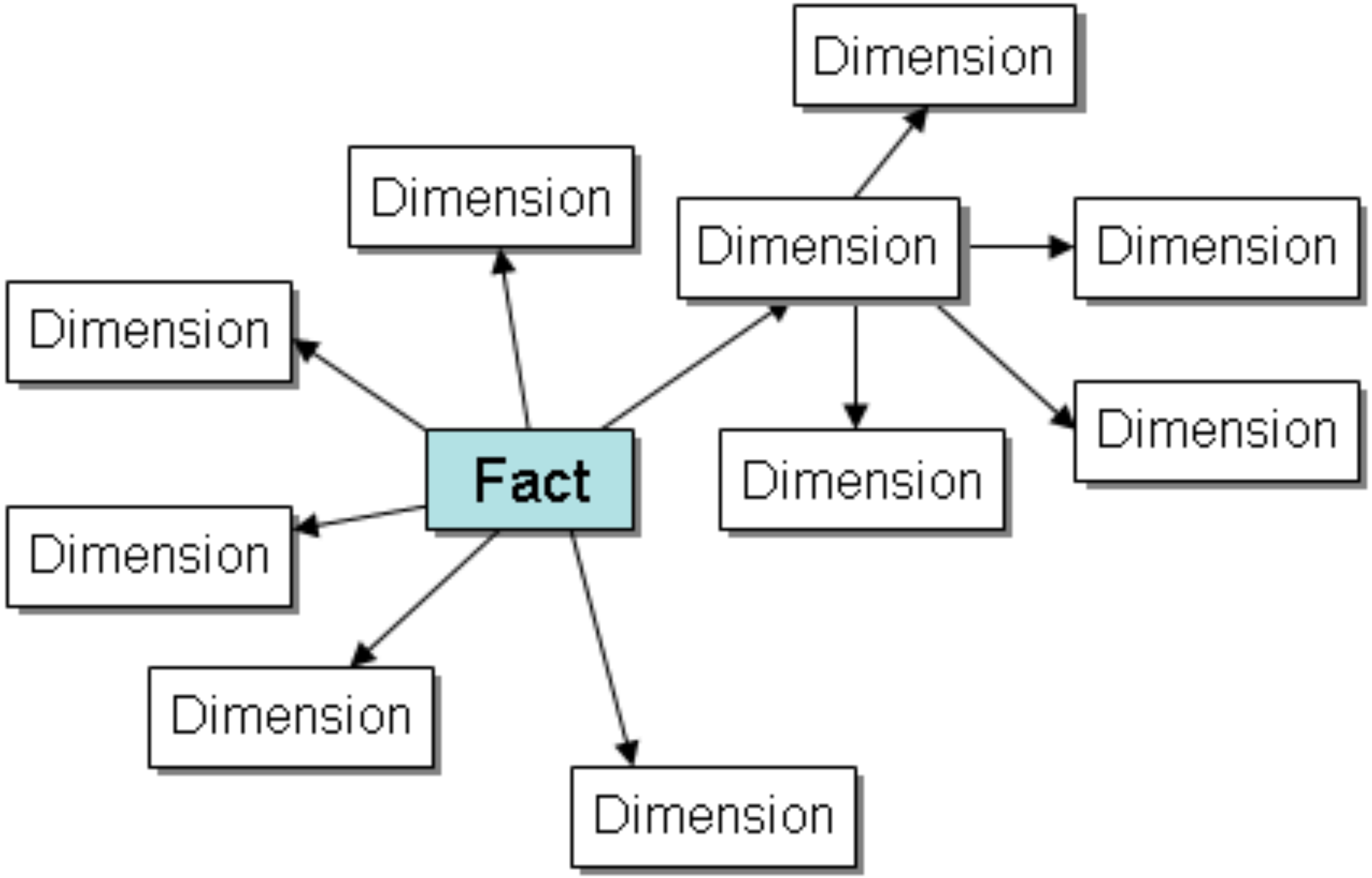
promotion_sk	name	ad_type	coupon_type
18	New Year sale	Poster	NULL
19	Aquarium deal	Direct mail	Leaflet
20	Coffee & cake bundle	In-store sign	NULL

Snowflake Schema

Greater normalization increases complexity, and requires more joins

BUT:

Normalization reduces redundancy and (therefore) storage requirements.



Columnar Storage

Data from each column is stored contiguously on disk. Aggregates on particular columns require less disk I/O.

For example: parquet

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