Storage and Retrieval

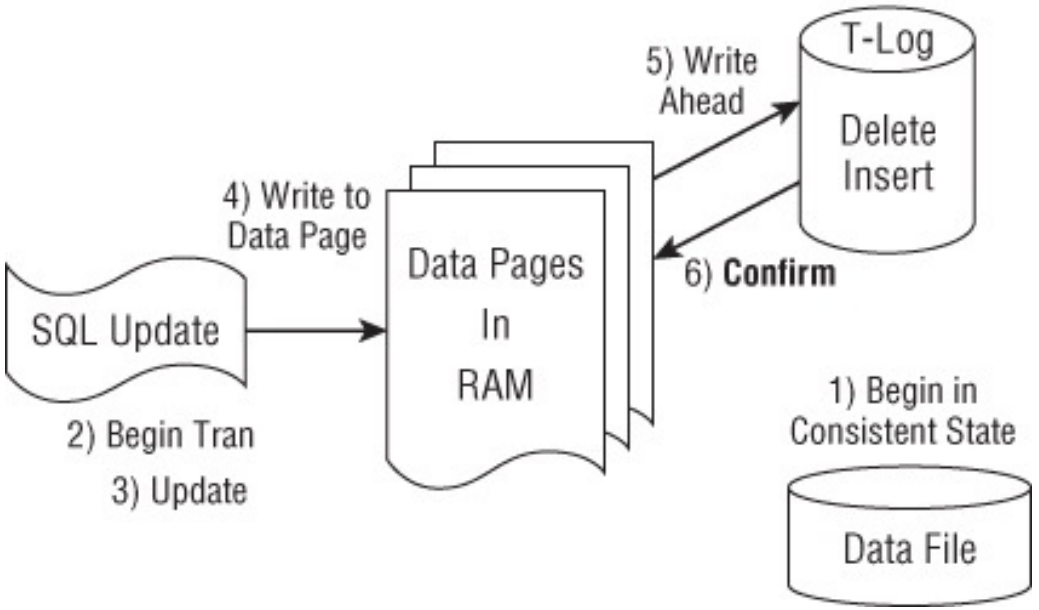
UA.DETI.CBD
José Luis Oliveira / Carlos Costa

Objectives

- How we can store the data.
- How we can find it again.
- How to select a storage engine that is appropriate for an application, from the many that are available.
- Difference between storage engines that are optimized for transactional workloads and those that are optimized for analytics.

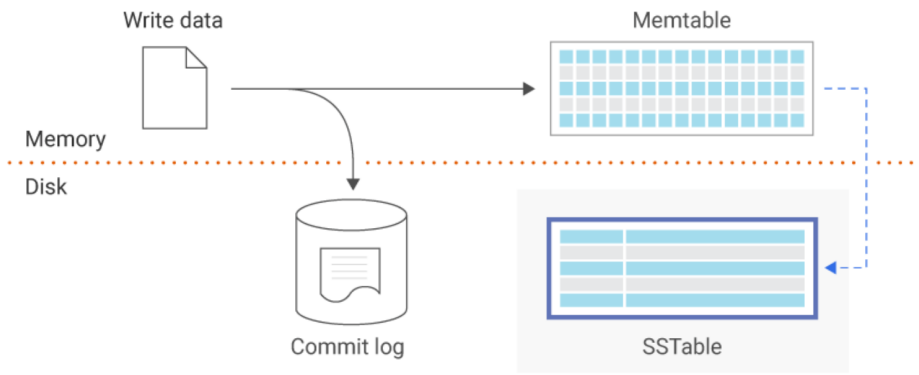
Big Picture – RDMS

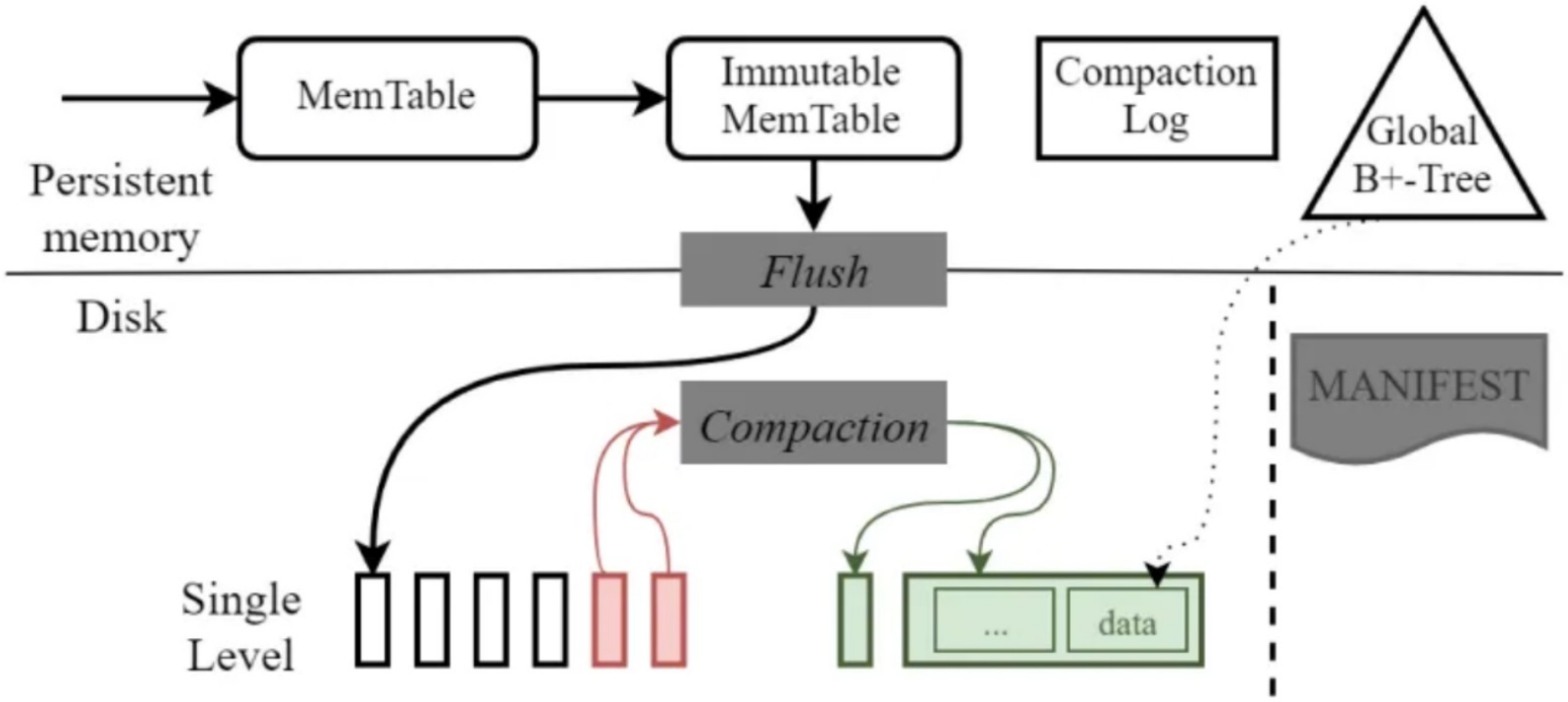
- Heap
  - Flat structure: File_ID | Page_ID | Row
    - Append rows to page
    - File contains pages
- B-Tree
  - Secondary Indexes
  - Clustered
- Transaction Log
  - Append-only log



Big Picture – NoSQL Engines

- Log-Structured Merge Tree
  - Commit-log (write-ahead log)
  - Memtable (sorted k-v@memory)
  - SSTable (sorted k-v@disk)
    - Compaction procedure
    - Indexing (e.g. B-tree)





Data Structures – Bash example

- Consider a simple key-value store:

```
#!/bin/bash

cbd_set () {
    echo "$1,$2" >> database
}

cbd_get () {
    grep "^$1," database | sed -e "s/^$1,//" | tail -n 1
}
```

- The key and value can be (almost) anything you like (e.g. the value could be a JSON document).
- Example:

```
$ cbd_set 123456 '{"name":"London","attractions":["Big Ben","London Eye"]}'
$ cbd_set 42 '{"name":"San Francisco","attractions":["Golden Gate Bridge"]}'
$ cbd_get 42
{"name":"San Francisco","attractions":["Golden Gate Bridge"]}
```

Bash example

- This storage format is very simple
  - a text file where each line contains a key-value pair, separated by a comma
  - Every call to **cbd_set** appends to the end of the file
  - Old versions of the value are not overwritten

```
$ cbd_set 42 '{"name":"San Francisco","attractions":["Exploratorium"]}'

$ cbd_get 42
{"name":"San Francisco","attractions":["Exploratorium"]}

$ cat database
123456,{"name":"London","attractions":["Big Ben","London Eye"]}
42,{"name":"San Francisco","attractions":["Golden Gate Bridge"]}
42,{"name":"San Francisco","attractions":["Exploratorium"]}
```

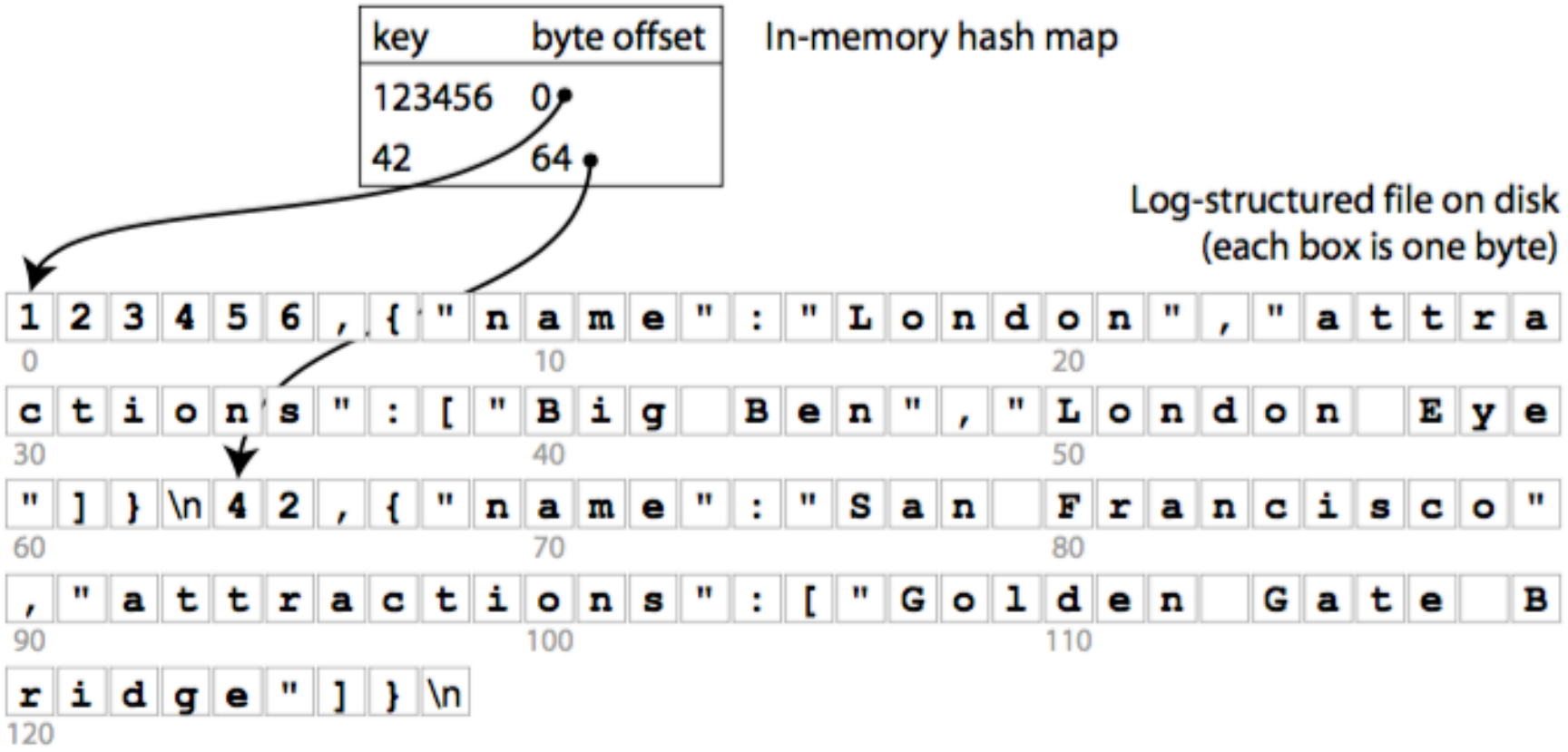
Bash example – performance

- The `cbd_set` function has a good performance for something that is so simple
  - appending to a file is generally very efficient.
  - Many databases use an append-only data file or logs.
- But, as the DB grows, the `cbd_get` function has a bad performance - *O(n)*.
- To efficiently find the value for a key, we need a different data structure: an index.
  - An index is an additional structure that is derived from the primary data.
  - Well-chosen indexes speed up read queries, but every index slows down writes.

Hash indexes

- Key-value stores are like a *dictionary* which is usually implemented as a hash map.
- A simple indexing strategy: keep an in-memory hash map where every key is mapped to a byte offset in the data file.
- This is essentially what some key-value databases do (e.g. Bitcask/Riak)
  - they offer high-performance reads and writes, if the hash map is kept completely in memory.

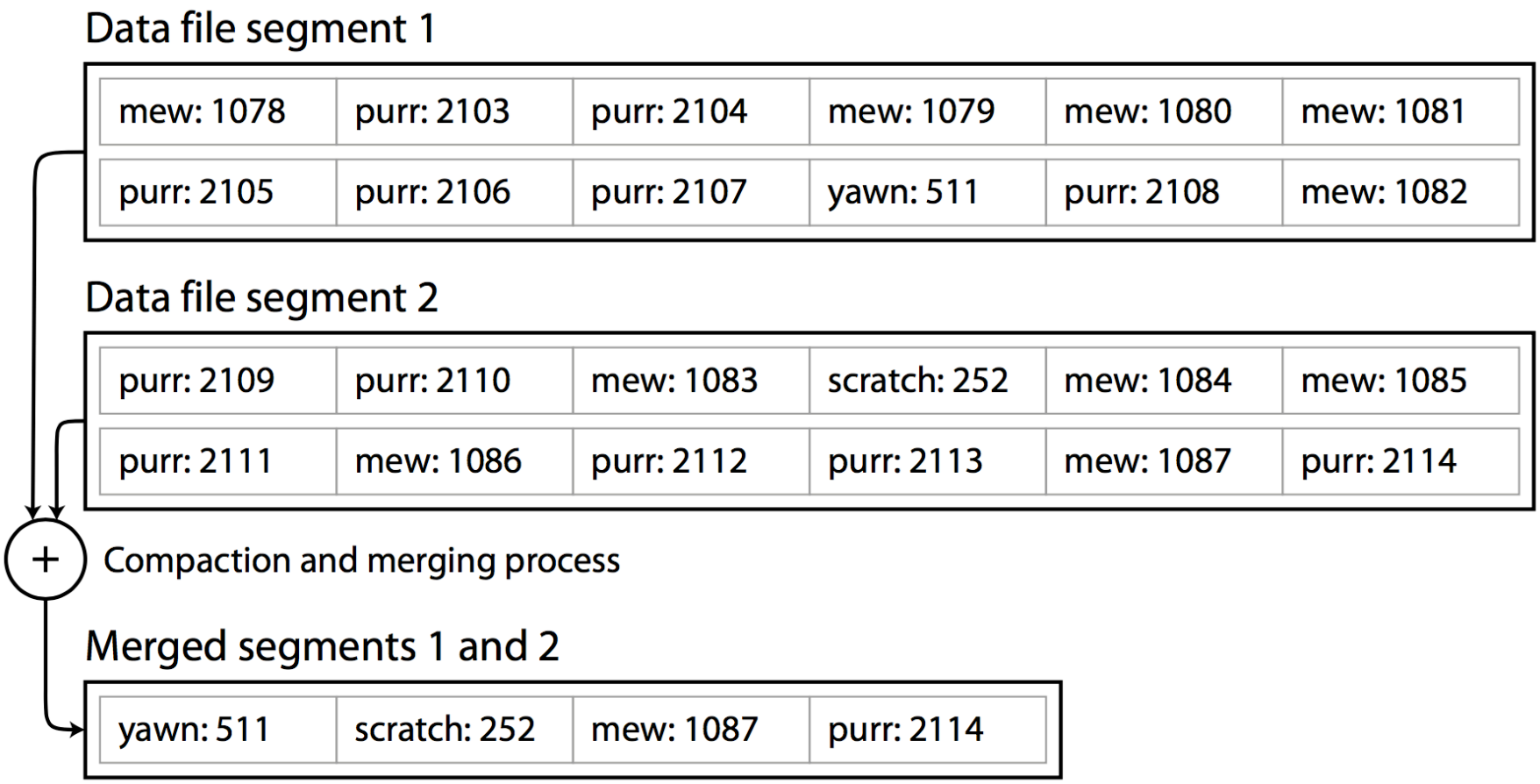
Hash indexes



Managing disk space

- How do we avoid running out of disk space?
  - Creating segments (for better performance).
  - Each segment contains all the values written to the database during some period of time.
  - Performing compaction (throwing away duplicate keys in the log).
- The merging and compaction can be done in a background thread.
  - We can continue to serve read and write requests as normal, using the old segment files.

Compaction and merging



Other issues

- File format
  - CSV is not the best format for a log.
  - A binary format may be used here.
- Deleting records
  - To delete a key, we need to append a special deletion record to the data file (tombstone).
  - When log segments are merged, the process discards any previous values for the deleted key.
- Crash recovery
  - If the system is restarted, the in-memory hash maps are lost.
  - To speed up recovery, we may store a snapshot of each segment's hash map on disk.

Other issues

- Partially written records
  - The database may crash at any time, including halfway through appending a record to the log.
  - Checksums may allow such corrupted parts of the log to be detected and ignored.
- Concurrency control
  - As writes are appended in sequential order, we may have only one writer thread.
  - Data file segments are append-only and immutable, so they can be concurrently read by multiple threads.

Append-only log

- Append-only design seems wasteful at first glance.
  - why don't you update the file in place, overwriting the old value with the new value?
- But, it turns out to be good for several reasons:
  - Appending and segment merging are sequential write operations, which are generally much faster than random writes.
  - Concurrency and crash recovery are much simpler if segment files are append-only or immutable.
- However, it also has limitations:
  - The hash table must fit in memory. It is difficult to make an on-disk hash map perform well.
  - Range queries are not efficient. For example, search all keys between A01 and A99.

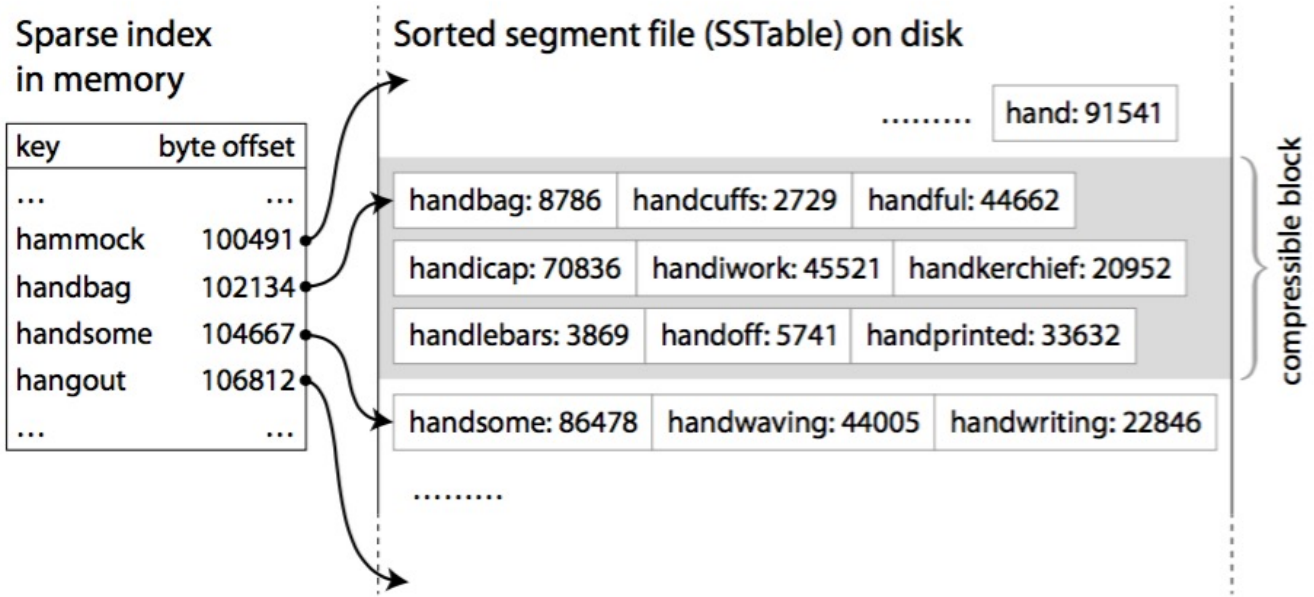
Sorted String Table (SSTable)

- In the previous examples, the key-value pairs appear in the order that they were written.
  - But, we can assure that the sequence of key-value pairs is *sorted by key*.
  - At first glance, that requirement seems to break our ability to use sequential writes.

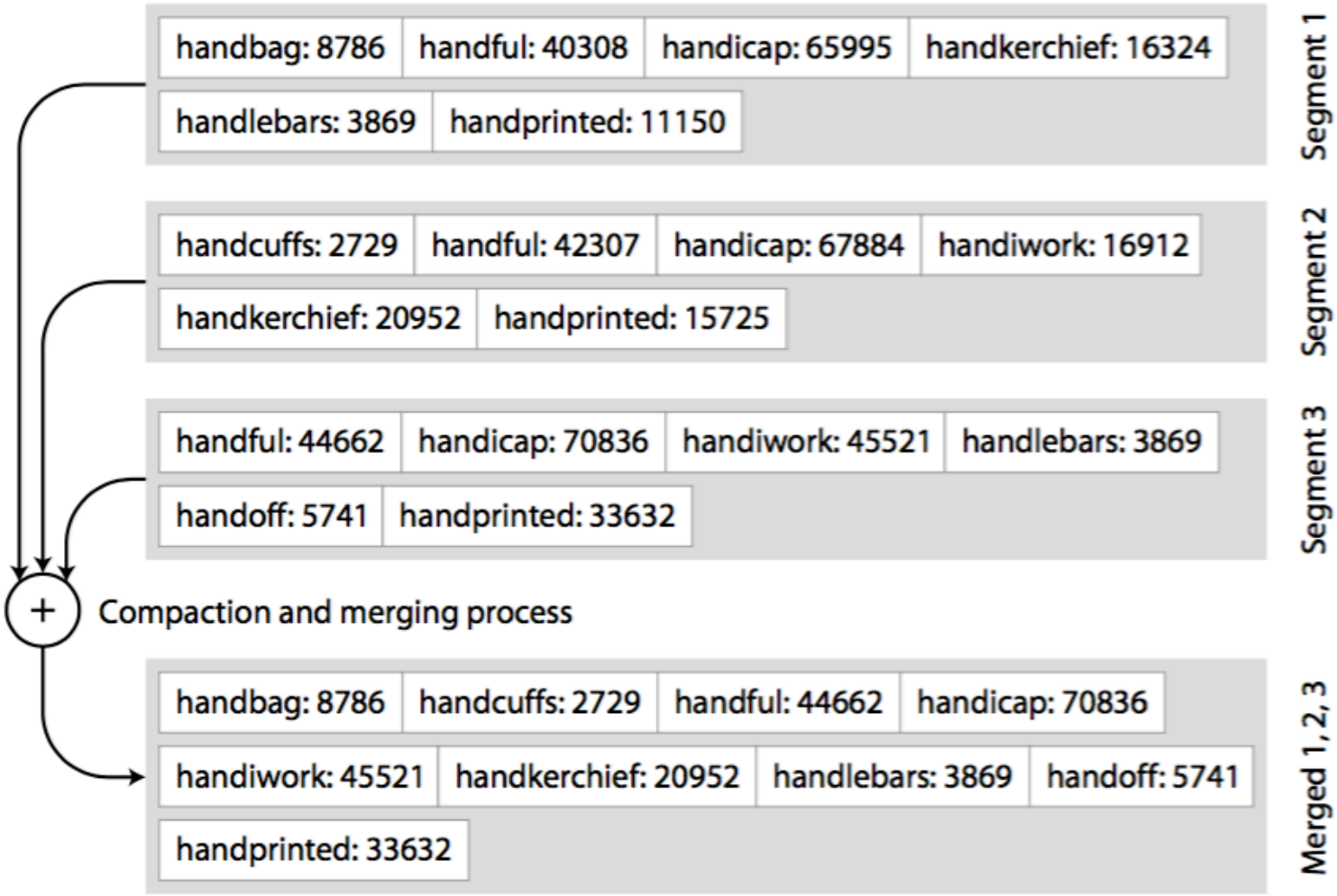
- We can use *Sorted String Table*, or *SSTable* for short.
- We also require that each key only appears once within each merged segment file (the merging process already ensures that).

SSTable

- SSTables have several advantages over log segments with hash indexes:
  - Merging segments is simple and efficient, even if the files are bigger than the available memory (like the mergesort algorithm).
  - No need to keep an index of all the keys in memory (e.g. just "aa", "ab", "ac", ...).



Merging SSTable segments



SSTable – sorting

- How do you sort the data by key?
  - Incoming writes can occur in any order.
- We can maintain a sorted structure in memory.
  - Using tree data structures (B-trees, AVL, Red-Black trees, …).
  - We can insert keys in any order, and read them back in sorted order.
  - This in-memory tree is sometimes called a *memtable*.

SSTable – sorting

- When the memtable gets bigger than some threshold, write it out to disk as an SSTable file.
  - This can be done efficiently because the tree already maintains the key-value pairs sorted by key.
  - The new SSTable file becomes the most recent segment of the database.
  - When the new SSTable is ready, the memtable can be emptied.
- To serve a read request:
  - First search the key in the memtable, then in the most recent on-disk segment, then in the next-older segment, etc.
- From time to time, run a merging and compaction process in the background to combine segment files and to discard overwritten or deleted values.

SSTable and Log

- This scheme suffers from one problem:
  - if the database crashes, the most recent writes (which are in the memtable but not yet written out to disk) are lost.
- To avoid that problem, we can keep a separate log on disk to which every write is immediately appended.
  - Every time the memtable is written out to an SSTable, the corresponding log can be discarded.

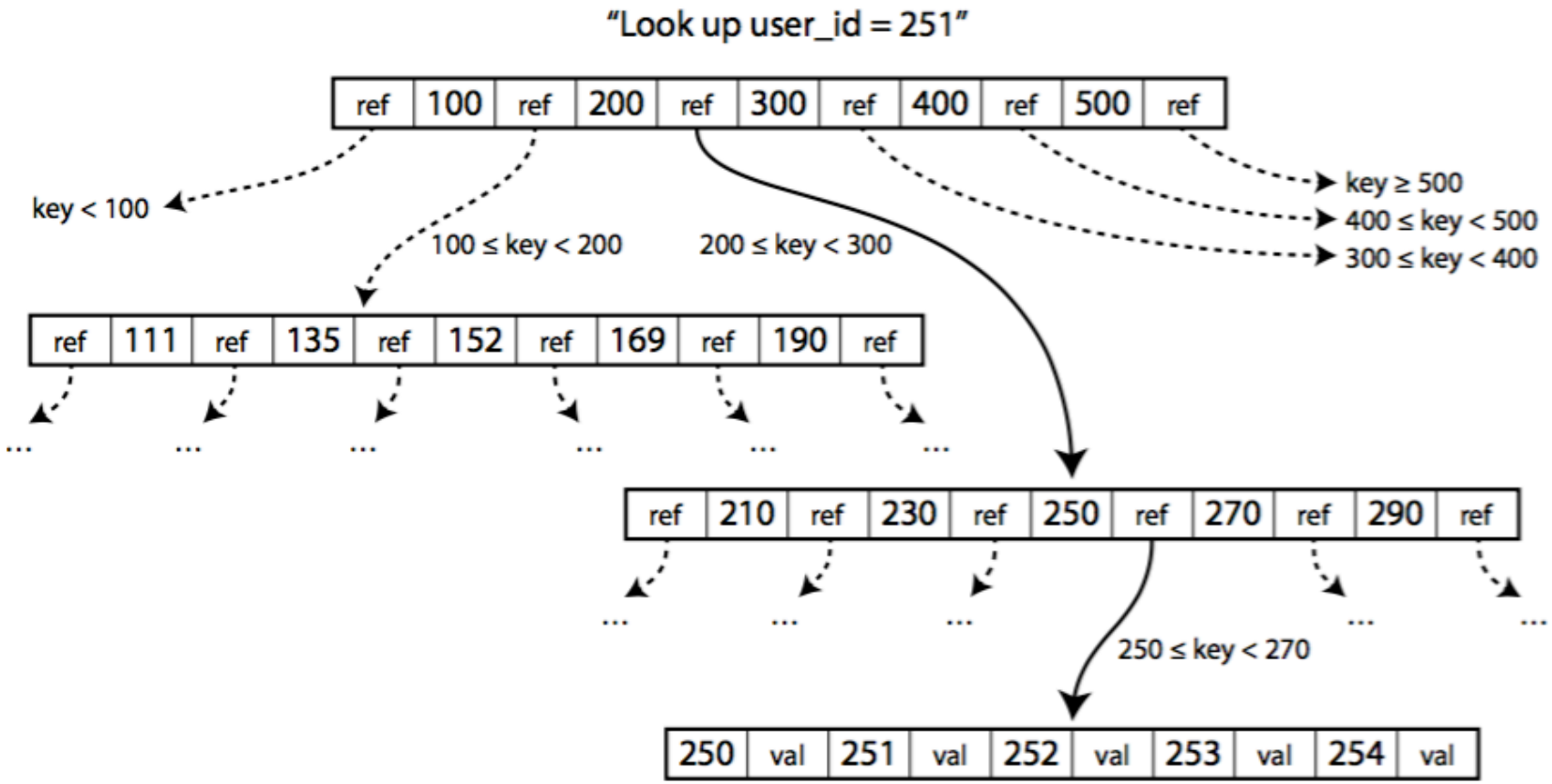
SSTable applications

- Google's Bigtable
  - which introduced the terms *SSTable* and *memtable*.
- LevelDB and RocksDB
  - key-value storage engine libraries that are designed to be embedded into other applications (e.g. Riak).
- Cassandra and HBase
  - both inspired by Google's Bigtable.
- Lucene
  - an indexing engine for full-text search used by Elasticsearch and Solr.

B-trees

- B-tree are the standard index implementation in almost all relational databases and many non-relational databases.
- Like SSTables, B-trees keep key-value pairs sorted by key, which allows efficient key-value lookups and range queries.
- B-trees break the database down into fixed-size blocks or pages, traditionally 4 kB in size, and read or write one page at a time.
  - This corresponds more closely to the underlying hardware, as disks are also arranged in fixed-size blocks.

B-trees


"Look up user_id = 251"
ref 100 ref 200 ref 300 ref 400 ref 500 ref
key < 100
100 ≤ key < 200
200 ≤ key < 300
300 ≤ key < 400
400 ≤ key < 500
key ≥ 500
ref 111 ref 135 ref 152 ref 169 ref 190 ref
ref 210 ref 230 ref 250 ref 270 ref 290 ref
250 ≤ key < 270
250 val 251 val 252 val 253 val 254 val

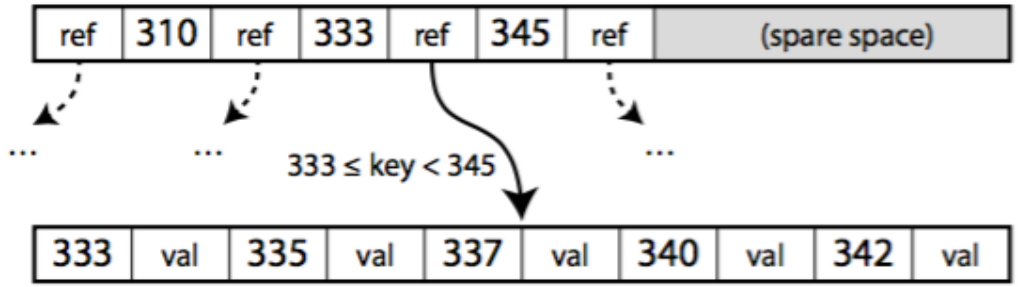
B-trees

- The structure starts at the root page.
- Each page contains *k* keys and *k + 1* references to child pages
  - *k* would typically be in the hundreds.
- Each child is responsible for a continuous range of keys, and the keys in the root page indicate where the boundaries between those ranges lie.

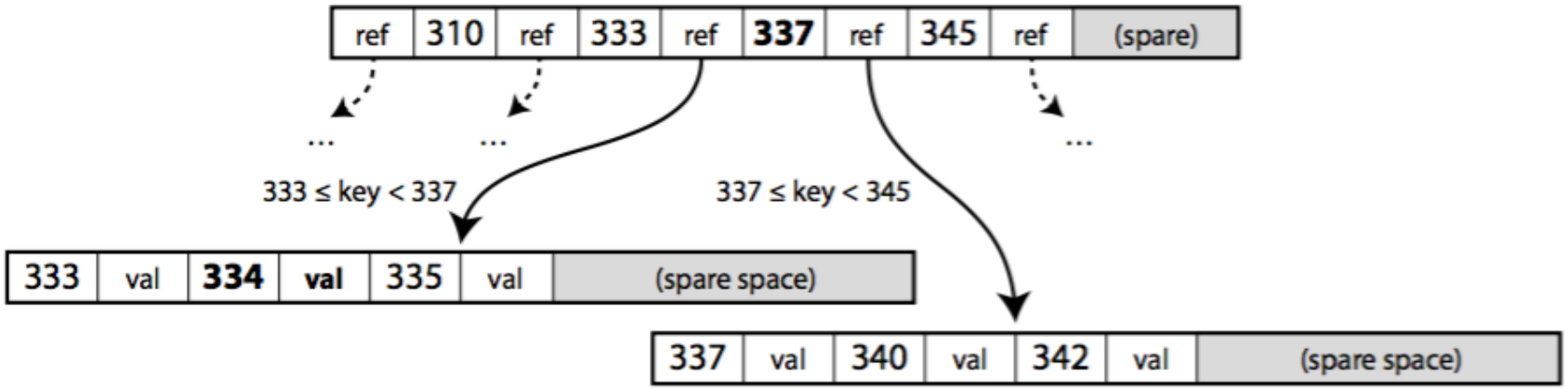
B-trees – operations

- Adding a new key:
  - find the page whose range encompasses the new key, and add it to that page.
  - If there isn't enough free space in the page to accommodate the new key, it is split into two half-full pages, and the parent page is updated to account for the new subdivision of key ranges.
- Updating the value for an existing key:
  - search for the leaf page containing that key, and change the value that page, and write the page back to disk (any references to that page remain valid).
- The tree remains *balanced*
- Time complexity (Search, Insert, Delete)
  - $O(\log_k n)$ for a B-tree with k entries per block and n keys

B-tree – splitting a page



After adding key 334:



Update-in-place vs. append-only log

- The basic underlying write operation of a B-tree is to overwrite a page on disk with new data.
  - Log-structured indexes only append to files but never modify files in place.
- Some operations require several different pages to be overwritten.
- If there is a crash after writing only some of the pages, we end up with a corrupted index.
  - e.g. an orphan page which is not a child of any parent.
- To deal with crashes, it is normal to include a *write-ahead log* (WAL, or *redo log*).
  - An append-only file to which every B-tree modification must be written before it can be applied to the tree.

Update-in-place vs. append-only log

- An additional complication of updating pages in-place is that careful concurrency control is required.
  - if multiple threads are going to access the B-tree at the same time, a thread may see the tree in an inconsistent state.
- This is typically done by protecting the tree's data structures with latches (lightweight locks).
- In this case, log-structured approaches are simpler
  - The merging and swapping occur in background without interfering with incoming queries.

B-tree optimizations

- Copy-on-write scheme.
  - To deal with crash recovery, a modified page is written to a different location, and a new version of parent pages in the tree is created, pointing at the new location.

- We can save space in pages by not storing the entire key, but abbreviating it.
  - Packing more keys into a page allows the tree to have a higher branching factor, and thus fewer levels.

- B-tree variants such as fractal trees borrow some log-structured ideas to reduce disk seeks.

B-tree versus LSM-trees

- LSM-trees are typically faster for writes, whereas B-trees are thought to be faster for reads.
- A downside of log-structured storage is that the compaction process can sometimes interfere with the performance of ongoing reads and writes.
- In B-trees, each key exists in exactly one place in the index, whereas a log-structured storage may have multiple copies in different segments.
  - This makes B-trees attractive in databases that want to offer strong transactional semantics.
  - In many relational databases, transaction isolation is implemented using locks on ranges of keys, and in a B-tree index, those locks can be directly attached to the tree.

Other indexing structures

- It is also very common to have secondary indexes.
  - The main difference is that keys are not unique, i.e. there might be many rows (documents, vertices) with the same key.
- This can be solved two ways:
  - either by making each value in the index a list of matching row identifiers (like a posting list in a full-text index), or
  - by making each key unique by appending a row identifier to it.
- Both B-trees and log-structured indexes can be used as secondary indexes.

Storing values within the index

- The key is the thing that queries search for, but the value could be one of two things:
  - the actual row (document, vertex) in question,
  - a reference to the row stored elsewhere.
- In the reference case, the place where rows are stored is known as a ***heap file***.
  - It avoids duplicating data when multiple secondary indexes are present.
- When updating a value, the heap file approach can be quite efficient.
- The record can be overwritten in-place, if the new value is not larger than the old value.

Storing values within the index

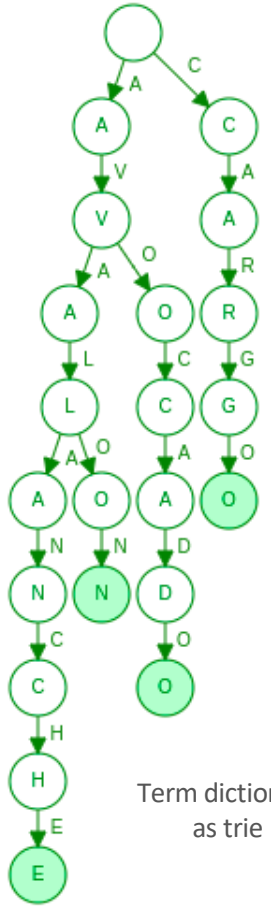
- In some situations, it can be desirable to store the indexed row directly within an index. This is known as a ***clustered index***.
  - The primary key of a table can be a clustered index, and secondary indexes refer to the primary key (rather than a heap file location).
- A compromise between a clustered and a nonclustered index is known as a ***covering index***.
  - It stores some table's columns within the index.
- These indexes can speed up reads, but
  - They require additional storage and overhead on writes.
  - Databases also need additional effort to enforce transactional guarantees, because of the duplication.

Multi-column indexes

- The indexes discussed so far only map a single key to a value.
  - That is not sufficient if we need to query multiple columns of a table (or multiple fields in a document) simultaneously.
- The most common type of multi-column index is called a concatenated index.
  - Combines several fields into one key by appending one column to another.
- This is like a phone book, which provides an index from (*lastname, firstname*) to phone number.
  - This index can be used to find all the people with a particular last name, or all the people with a particular lastname-firstname combination.

Fuzzy indexes

- All the indexes discussed so far assume that we query for exact values of a key, or a range of values of a key with a sort order.
  - How to search for similar keys, such as misspelled words.
- Some data stores (e.g. Lucene) allow searching text within a certain edit distance.
  - Lucene uses a SSTable-like structure for its term dictionary.
  - This structure tells queries at which offset in the sorted file they need to look for a key.
  - This is finite state automaton over the characters in the keys, like a trie, which supports efficient search for words within a given edit distance.



Term dictionary as trie

Keeping everything in memory

- For many datasets, it is feasible to keep them entirely in memory.
  - Potentially distributed across several machines.
- This led the development of in-memory databases.
- Some in-memory key-value stores, such as Memcached, are intended for caching use only.
  - Acceptable for data to be lost if a machine is restarted.
- But others aim for durability.
  - by writing a log of changes to disk, by writing periodic snapshots to disk, or by replicating the in-memory state to other machines.
  - Writing to disk also has operational advantages: files on disk can easily be backed up, inspected and analyzed by external utilities.

In-memory solutions

- VoltDB, MemSQL and Oracle TimesTen are in-memory databases with a relational model.
- RAMCloud is an open-source in-memory key-value store with durability (using a log-structured approach for the data in memory as well as the data on disk).
- Redis and Couchbase provide weak durability by writing to disk asynchronously.

In-memory solutions

- The performance advantage of in-memory databases is not due to the fact that they don't need to read from disk.
  - Even a disk-based storage engine may never need to read from disk if you have enough memory.
- Rather, they can be faster because they can avoid the **overheads of encoding in-memory** data structures in a form that can be written to disk.
  - Besides performance, they provide data models that are difficult to implement with disk-based indexes.
  - For example, Redis offers a database-like interface to various data structures such as priority queues and sets.
  - By keeping all data in memory, its implementation is comparatively simple.

Transaction Processing and Transaction Analytics

Online Transaction Processing (OLTP)

- In the early days, a write to the database typically corresponded to a commercial **transaction**:
  - making a sale, placing an order with a supplier, paying an employee's salary, etc.
- Despite databases expanding into many other areas, the term *transaction* nevertheless stuck.
  - ***Transaction processing*** means allowing clients to make low-latency reads and writes.
  - On the other hand, ***batch processing*** jobs run periodically, for example once per day.
- Applications are typically interactive (insert, update, search, etc.).
  - This access pattern became known as ***online transaction processing*** (**OLTP**).

Data analytics

- Databases also started being increasingly used for *data analytics*.
  - which has very different access patterns than OLTP.
- Usually, an analytic query needs to scan over a huge number of records and calculates aggregate statistics.
- Some examples:
  - What was the total revenue of each of our stores in January?
  - How much more bananas than usual did we sell during our latest promotion?
  - Which brand of baby food is most often purchased together with brand X diapers?

Online Analytic Processing (OLAP)

- Analytic queries are often written by business analysts.
  - and feed into reports that help a company's management make better decisions (business intelligence).
- To differentiate this pattern of using databases from transaction processing, this has been called

  online analytic processing (**OLAP**)

OLTP versus OLAP

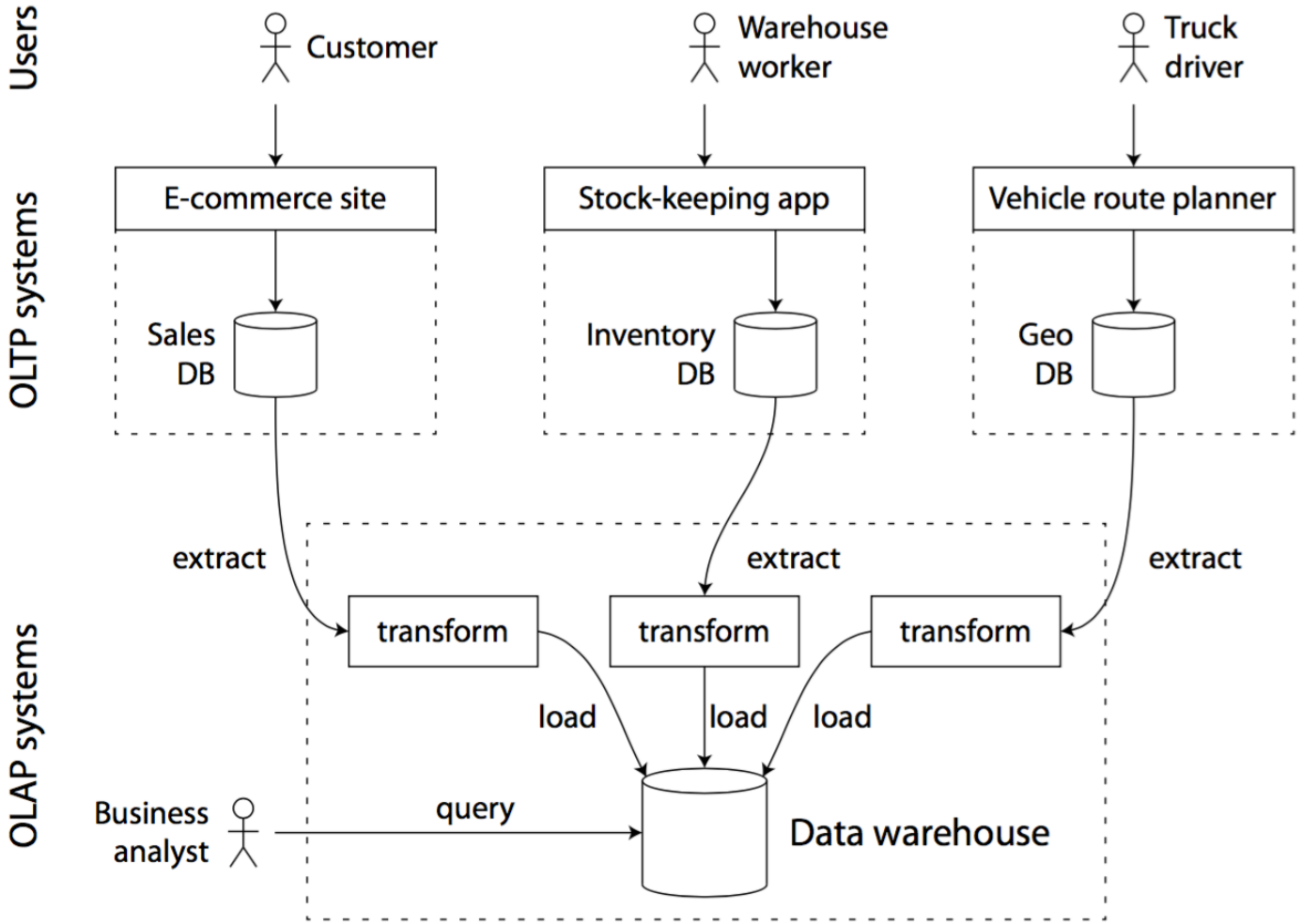
Property	OLTP	OLAP
Main read pattern	Small number of records per query, fetched by key	Aggregate over large number of records
Main write pattern	Random-access, low-latency writes from user input	Bulk import (ETL) or event stream
Primarily used by	End user/customer, via web application	Internal analyst,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

- RDBMS and SQL work well for OLTP-type queries as well as OLAP-type queries.
- However, OLAP normally uses a separate database from OLTP, i.e a **data warehouse**.

Data warehousing

- **OLTP** systems are expected to be **highly available** and to process transactions with **low latency**.
- A **data warehouse** is a **separate database** that analysts can query without affecting OLTP operations.
  - It typically contains a **read-only copy** of the **data** in all the various OLTP systems in the company.
- Data is extracted from OLTP databases, transformed, cleaned, and loaded into the data warehouse.
  - This process of getting data into the warehouse is called ***Extract-Transform-Load*** (ETL).

Data warehousing – ETL



Why Separate Data Warehouse?

- Why combine OLTP and OLAP?
  - Data warehouses are commonly relational because SQL fits for analytic queries well.
  - Many OLTP tools (querying, visualization, etc.).
- Why separate?
  - Different functions and different data:
    - missing data: Decision support requires historical data which operational DBs do not typically maintain.
    - data consolidation: Decision support requires data consolidation (aggregation, summarization) from heterogeneous sources.
    - data quality: different sources typically use inconsistent data representations, codes, and formats which must be reconciled.
- Many vendors now support either transaction processing or analytics workloads.

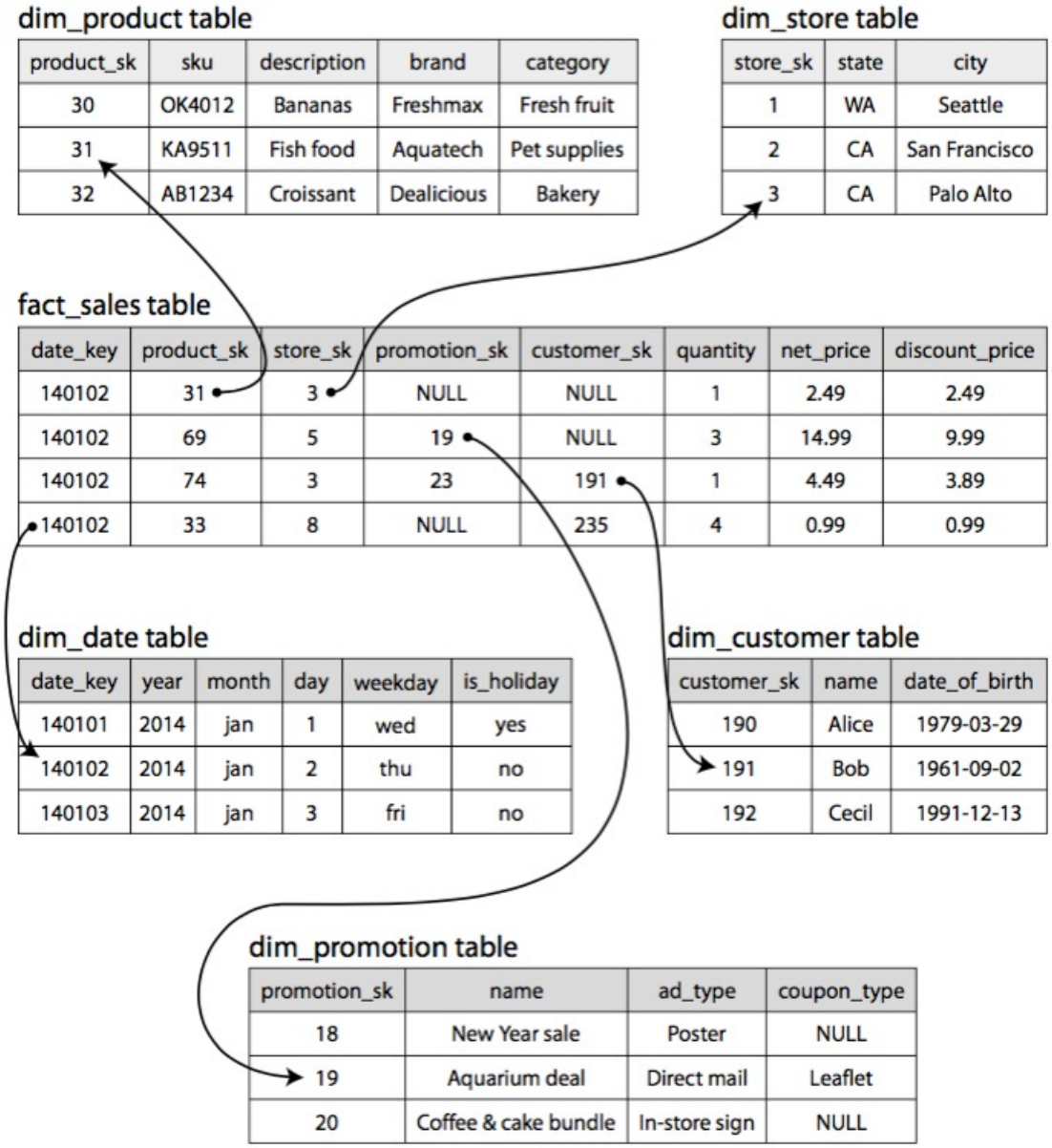
Data Warehouses: some solutions

- Commercial
  - Amazon Redshift, IBM PureData, MS SQL Parallel Data Warehouse, Oracle Exadata, SAP Business Warehouse, Teradata, Vertica.
- Open source
  - Apache Hive, AMPLab's Shark, Cloudera Impala, Hortonworks Stinger, Facebook Presto, Apache Tajo, Apache Drill.

Schemas for analytics

- Many data warehouses use a ***star schema*** (or dimensional modeling).
- The main entity is the ***fact table***.
  - Each fact table row represents an event at a particular time (a purchase, a page view, etc.).
  - Some columns are attributes (e.g. price). Others are references (foreign keys) to other tables (**dimension tables**).
- A variation of the star template is the ***snowflake schema***
  - The dimensions are broken down into sub-dimensions (more normalized but more complex to work with).

Schemas for analytics – Example



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

Querying problem

- A fact table is typically huge – can have trillions of rows and petabytes of data.
  - Storing and querying becomes a challenging problem.
- Fact tables are often over 100 columns wide.
- But, a typical data warehouse query only accesses a few of them at one time.

- Problem?

- Solution?

Querying problem

- Problem:
  - We need to read the entire table to process one single column.
  - OLTP databases are typically row-oriented: all the values from one row of a table are stored next to each other.

- Solution:
  - ***Column-oriented storage***

Column-oriented storage

- The idea:
  - don't store all the values from one row together, but store all the values from each column together instead.
  - If each column is stored in a separate file, a query only needs to read and parse those columns.
- The column-oriented storage layout relies on each column containing the rows in the same order.
- To reassemble an entire row *n*, we need to take all the *n* entries from each column file.

Column-oriented storage

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

Column compression

- Column-oriented storage often lends itself very well to compression.
- The sequences of values for each column are often repetitive.
- Different compression techniques can be used depending on the data in the column.
  - Bitmap encoding: technique particularly effective in data warehouses
    - One binary vector per every term in the column
  - Run Length Encoding (RLE)
    - TSLA, TSLA, TSLA, TSLA, TSLA, SQ, SQ, SQ, APPL, AAPL **=> TSLA x 5, SQ x 3, AAPL x 2**

Column sorting

- Normally, columns are stored in the inserted order.
- However, we can choose another order, using one-column values.
  - It serves as an indexing mechanism.
- A second column can determine the sort order of any rows that have the same value in the first sorting column.
  - E.g., if **date_key** is the first sort key, and **product_sk** the second, all sales for the same product on the same day are grouped together.
- Another advantage of sorted order is that it can help with compression of columns.

Writing problem

- Column-oriented storage, compression and sorting all help to make OLAP queries faster.
- **What about writing?**
- **What's the best structure?**

Writing problem

- Column-oriented storage, compression and sorting all help to make OLAP queries faster.

- **What about writing?**
  - As rows are identified by their position within a column, the insertion has to update all columns consistently.
- **What's the best structure?**

Writing to column-oriented storage

- A good solution: LSM-trees (Log Structured Merge)

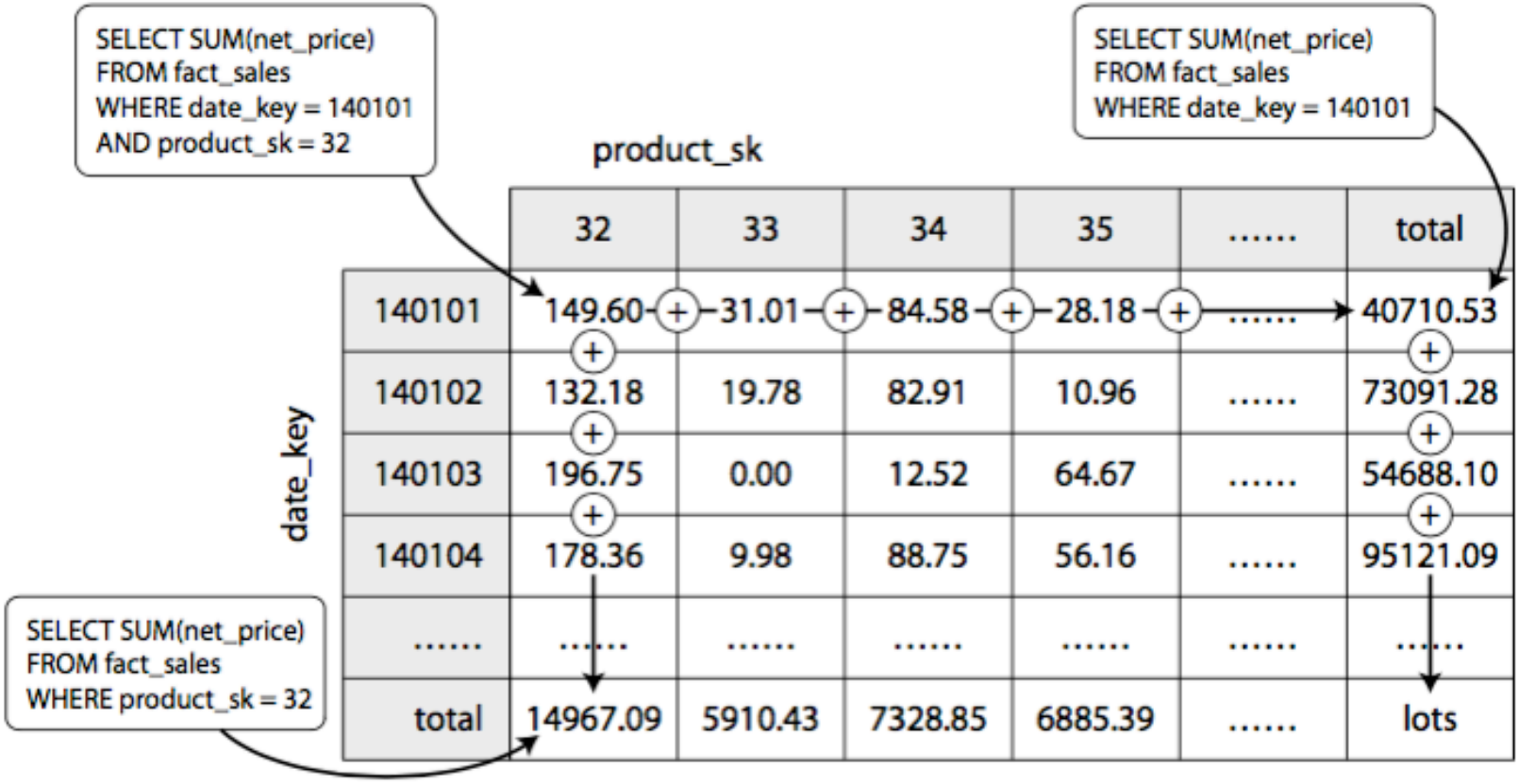
- All writes first go to an in-memory store, where they are added to a sorted structure, and prepared for writing to disk.
- When enough writes have accumulated, they are merged with the column files on disk, and written to new files in bulk.
- Queries need to examine both the column data on disk and in memory.

Materialized views

- Data warehouse queries often involve an aggregate function (COUNT, SUM, AVG, MIN, ...).
- Aggregates can be used by many different queries.
  - Repeating the data processing.
- Solution?
- Cache these aggregates in a ***materialized view***.
  - When the underlying data changes, a materialized view needs to be updated.
- A ***data cube*** or ***OLAP cube*** is a special case of a materialized view, which is a grid of aggregates grouped by different dimensions.

Data cube

- Example: Two-dimensional data cube, aggregating data by summing



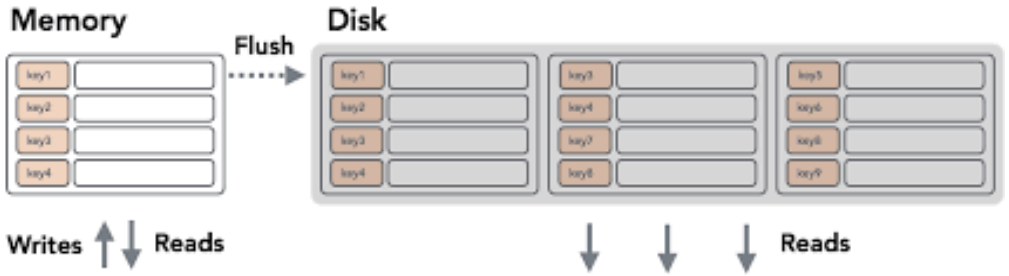
Summary

- Log-structured storage.
  - e.g. Append-only Log, Append-Log, Write-Ahead Log (WAL)

- Only allows appending to files and deleting obsolete files
  - Never updates a file that has been written.
- Enables higher write throughput.
- Read not .. so good.

Summary

- Log Structured **Merge Tree** storage (LSM-trees).
  - Buffered write, Multi-level storage
- They are immutable
  - SSTables are written on disk once and never updated.
  - Flushed tables can be accessed concurrently.
- They are write-optimized
  - writes are buffered and flushed on disk sequentially.
- Reads might require accessing multiple sources
- Compaction is required, as buffered writes are flushed on disk.



Summary

- Update-in-place storage
  - Indexing structures and databases that are optimized for keeping all data in memory.
  - Treats disk as fixed-size pages which can be overwritten.
- B-trees are the biggest example of this philosophy.
  - They are mutable, and do not require complete file rewrites or multisource merges.
  - They are read-optimized, i.e., do not require reading and merging from multiple sources.
  - Writes might trigger a cascade of node splits, making some write operations more expensive.
  - Concurrent access requires reader/writer isolation and involves chains of locks and latches.

Summary

- Storage engines fall into two broad categories:
  - optimized for transaction processing (OLTP).
  - optimized for analytics (OLAP).

- OLAP queries processing.
  - They require sequentially scanning of a large number of rows.
  - It is important to encode data very compactly, to minimize the amount of data that the query needs to read from disk.
- Column-oriented storage helps achieve this goal.

Resources

- Martin Kleppmann, ***Designing Data-Intensive Applications***, O'Reilly Media, Inc., 2017.
- Pramod J Sadalage and Martin Fowler, ***NoSQL Distilled*** Addison-Wesley, 2012.
- Eric Redmond, Jim R. Wilson. ***Seven databases in seven weeks***, Pragmatic Bookshelf, 2012.
- Hector Garcia-Molina, Jeffrey D. Ullman, Jennifer Widom, ***Database systems: the complete book*** *(2nd Ed.)*, Pearson Education, 2009.
- Petrov, Alex. "Algorithms Behind Modern Storage Systems." *Queue* 16, no. 2 (2018): 30.