**SYSTEM DESIGN NOTES (BOOK)**

Nowadays applications are data intensive and not compute intensive. Major concern is how manage complex large amount of data.

A data intensive application is build which have common functionality such as :

-> A data store such that other application can find it again. (DATABASES)

-> Remembering the result of expensive operations, to speed up reads. (CACHES)

-> Allow users to search data by keywords or filter in various ways (SEARCH INDEXES)

-> Send a message to another process, to be handled async (Stream Processing)

-> Periodically crunch a large of amount of accumulated data (Batch Processing)

In Data systems there are some core components which we want to take care mainly :

1. Reliability : The System should continue to work correctly (i.e performing correct functions at desired level of performance) even when adverse conditions occurs

Such as hardware or software faults or human error etc.

2. Scalability : As the System grows (in data volume, traffic or complexity) there should be reasonable ways to deal with the growth.

3. Maintainability : Over the period of time many people will work on the system( people from engineering or operations both maintaining current behaviour and

Adapting to new changes as well) and they should be able to work it on productively with time.

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

RELIABILITY

In general terms it means system should do well even when something bad happens, it should be fault tolerant. We can’t prevent every fault in the world but we

Try to make our system as much resilient as possible. There are some factors which we can make our system un-reliable and has to be dealt with :

-> Hardware Failures : One Reason for our system not being reliable can be some hardware failures are happening, for ex : Disk Failures, Faulty RAMs, Power Blackout,

Wrong network cable plugging, there can be various reasons for this. One way we solve this problem is using redundancy. That is

Having multiple disks, power backup, basically a redundant component which can take original one’s place if something wrong happens.

-> Software Failures : We usually think of hardware failures as random and independent from each other : one machine disk failing does not imply that other disk is also

going to fail. These cascading failures or multiple failures on hardware are rare. But there can be systematic error within the system. Such faults

are harder to anticipate because they are correlated across nodes and can cause many more failures. One bug can make one node corrupt and

other nodes may depend on that node and thus cascading failure may happen. There is no quick solution for this you have to think about logics,

assumptions being taken by the system, through testing, isolating and testing system etc etc.

-> Human Errors : Humans designs systems and it is human who runs them too. And we all know humans can make mistakes and can make a lot. So we have to build

the system in such a way that it is less prone to human error for ex: well designed APIs, admin interfaces make it easy to do the “right thing” and

discourage the “wrong behaviour”. However, if the interfaces are too restrictive people will work around them, negating their benefit, so

this is a tricky balance to get right. Decoupling the system components whenever required, through testing in isolated environments.

Allow easy recover y from human errors to minimize the impact of human error. Fast Roll back, pushing new code gradually so that

small subset of code affects less system and will easy to debug. Monitoring and metrics can help in identifying issues early.

 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

SCALABILITY

System is working correctly does not mean it will keep on working correctly in future also. With increase in load like concurrent users going from 1

To 100 million might degrade our system. So on scalability we should think like “If the system grows in a particular way, what are the options for

Coping with the growth ?” And “how can we add computing resources to handle addition load ?”

Describing Load : Loads can be described with few numbers which we call load parameters. The best choice of parameters depends on the type of architecture It can be request per second served to the web server, the ratio of reads to write in a DB, the numbers of simultaneous active users

in a chat room, the hit ratio on a cache etc.

Describing Performance : Once we have described the load, we can now investigate what will happen when load increases. There are two ways to look into it.

-> When we increase our load parameter and keep system resources(CPU, Memory, Bandwidth etc) unchanged, how performance

of system affected.

-> when we increase the load, how much do you need to increase the resources so that performance remains unchanged.

Coping with Load : People ofter dichotomy between horizontal and vertical scaling up, In horizontal scaling we increase the no. of resources and distribute

load, whereas in vertical scaling we increase the power of existing system. Working on a single machine is comparatively easier

but after a certain limit, it can be way costlier. Generally in reality we often see mixture of both, where multiple machine which are

reasonably fast are worked upon together to meet the load.

Some systems are elastic meaning they can increase or decrease the computing resources according to load change, whereas in some

system it is done manually. An elastic system can be useful if load is highly unpredictable, but manually scaled systems are simpler

and may have fewer operational surprises.

Keep in mind that, there is no specific formula that after that much load or complexity we should distribute our system in this way.

It depends on which type of load, which type of operations are most common etc.

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

MAINTAINABILITY

Majority of the cost of software is not in its initial dev. Stage but it is in its ongoing maintenance like fixing bugs, keeping it operational, adapting to new

Platforms, modifying for new use cases, adding features etc. In future we don’t want to face a situation where we have a legacy code where most of things

Are outdated and non-operational. So we will pay particular attention to three design principles for software systems:

*Operability :* Make it easy for operations teams to keep the system running smoothly.

Operations teams are vital to keeping a software system running smoothly. A good operations team typically is responsible for the following, and more :

* Monitoring the health of the system and quickly restoring service if it goes into a bad state.
* Tracking down the cause of problems, such as system failures or degraded performance
* Keeping software and platforms up to date, including security patches
* Keeping tabs on how different systems affect each other, so that a problematic change can be avoided before it causes damage  and many more.

*Simplicity :* Make it easy for new engineers to understand the system, by removing as much complexity as possible from the system.

*Evolvability* : Make it easy for engineers to make changes to the system in the future, adapting it for unanticipated use cases as requirements change.

Also known as *extensibility*, *modifiability*, or *plasticity*.

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

RELATIONAL MODEL VS DOCUMENT MODEL

In 1970s SQL Model appeared and dominated over 30-40 years, in-between many other models like hierarchical and object model came but vanished quickly. In around 2010s

NoSQL model came into existence and was adopted due to

-> A need for greater scalability than relational databases including very large datasets or very high write throughput.

-> A preference for free and open source software over commercial database products.

-> Specialized query operations that were not supported by relational model.

-> Desire for dynamic and expressive data model

Relational DB are better in handling relationships such as many-to-many and one-to-many, they can simply put reference of other entities in terms of ids

And can do joins and foreign reference and handle it efficiently.

Whereas these kinds of operations are either not possible or very costly in non-relational DB, So if data is not so much dynamic and closely

coupled then it would be better to go with relational DB.

-> The main argument in favour of document data model are schema flexibility, better performance due to locality, and customised data structures

According to the application needs. But Relational model counters it by providing better support for joins, many to many and many to one relationships.

-> If the data in our application has document like structure(like that linked-in profile example: i.e., a tree of one-to- many relationships, where typically the entire tree is

loaded at once ) then its a good idea to use a document model. The relational technique of *shredding*— splitting a document-like structure into multiple tables

(like positions, education, and contact\_info)—can lead to cumbersome schemas and unnecessarily complicated application code.

-> When we have future changes in schema then document based model is more likeable choice before it can accommodate changes very easily, you can

Simply put new data in new format and it won’t complain. But in relational model you have specific format for storing and retrieving data. When writing

Into relational table a predetermined schema is there. And if requirements changes then table has to be altered and can result in downtime.

-> Data locality of queries is also very important factor, when you want to retrieve a complete document at once like rendering a website then, it makes

More sense to use non-relational db, rather than putting tables together and grouping all data for retrieval. The locality advantage only applies if you

need large parts of the document at the same time. The database typically needs to load the entire document, even if you access only a small portion

of it, which can be wasteful on large documents. On updates to a document, the entire document usually needs to be rewritten—only modifications

that don’t change the encoded size of a document can easily be performed in place. For these reasons, it is generally recommended that you keep

documents fairly small and avoid writes that increase the size of a document

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

**Graph-Like Data Models**

If application has mostly one to many relationship or no relationship between records then document model is appropriate but if applications has many-to-many

Relations very common. The relational model can handle simple cases of many-to-many relationships, but as the connections within your data become

more complex, it becomes more natural to start modelling your data as a graph.

A graph consists of two kinds of objects: *vertices* (also known as *nodes* or *entities*) and *edges* (also known as *relationships* or *arcs*)

There are three declarative query languages for graph : Cypher, SPARQL, and Datalog

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

**Storage and Retrievals**

We don’t need to know how all the storage engine are build piece to piece, but to understand which one you need for your application or how to tune it so it works best for

Your kind of data load, we should have rough idea about all these storage engines and they’re working under the hood.

Indexes : To speed up read we use indexing on DB, its kind of a index of book in the form of some metadata which helps in identifying the actual record quickly, but the

Tradeoff is that whenever we introduce indexing the write operation also slows down because we are not just simple appending data but according to indexes we have to

Now add or data. So, we need to choose suitable indexes for our DB which aligns to our required queries.

DATA STRUCTURES THAT POWERS OUR DB

HASH INDEXES :

We can hash offset to read quickly from DB.

For ex : let’s say we have a DB which stores data by simply appending to the last in the file, now we are storing the data in the form of key value pairs.

So, if we are just appending key value pairs one by one(without any space or newline), we can hash the keys and store offset with corresponding key.

So if we require the value of some ’x’ key, then we can simply look into our hash and see the offset and we can know from where the record start and we can read it.

We can merge the records overtime to remove duplicates and values which are expired and have new values.

But this look simple but works well, but in real world scenario, we may face some issues like :

File format (There are different file formats that we save), Deleting Records, Crash Recovery, Partially written records, concurrent control.

SSTables and LSM -Trees

-> We can make simple changes to format of our segment files : We require the sequence of key-value pair sorted by key. This format is SORTED STRING TABLE.

-> After merging also we should only have 1 key for a segment not duplicates. Merging of segment uses similar technique like merge sort, put all pointers on start of

Each segment and pick the smallest one among all segment pointers and move forward the pointer from the segment where value is fetched.

-> To find a particular value we don’t need to store offset of all the keys, but for a block of data we can store a key, then once we find that block we can find the data

Inside that block. This makes our stored key table sparse.

Constructing and maintaining SSTables

We can use RB-Tress or AVL Tress to insert record and read in sorted order.

We can make our storage engine work as follows :

-> When a write comes in add it to in-memory balanced tree data structure(AVL or RB Trees). This in-memory is sometimes called as memtable.

-> when the memtable gets bigger than some threshold we can store, we can write to disk as SSTable File.

-> In order to fulfil a read request first try to find the key in the memtable, then in the most recent on disk-segment, then int the next-older segment.

-> From time to time run a merging run a merging and compaction process, in the background to combine segment files to remove duplicate and deleted files.

It only has one drawback that if DB crashes then, it can lost current changes so, to fix it we can maintain a simple log file, which instantly append into hard disk, and is only

Purpose is to restore the segment.

Storage engines that are based on this principle of merging and compacting sorted files are often called LSM( Log Structured Merge ) storage engines.

Performance Optimizations : LSM tree can be slow when key does not exist, because it have to go deep level into every segment according to time added to ensure

That this record does not exists.In order to optimise this, storage engines often uses bloom filters.

Bloom Filters : A bloom filter is a memory efficient data structure for approximating the contents of a set. It can tell if a key does not exist in the database and saves

Many unnecessary calls.

There are also strategies to determine the order and timing of how SSTables are compacted and merged. The most common are size-tiered (newer and smaller SSTables are

Successively merged into older and larger SSTables)

and levelled compaction (In leveled compaction, the key range is split up into smaller SSTables and older data is moved into separate “levels,” which allows

the compaction to proceed more incrementally and use less disk space)

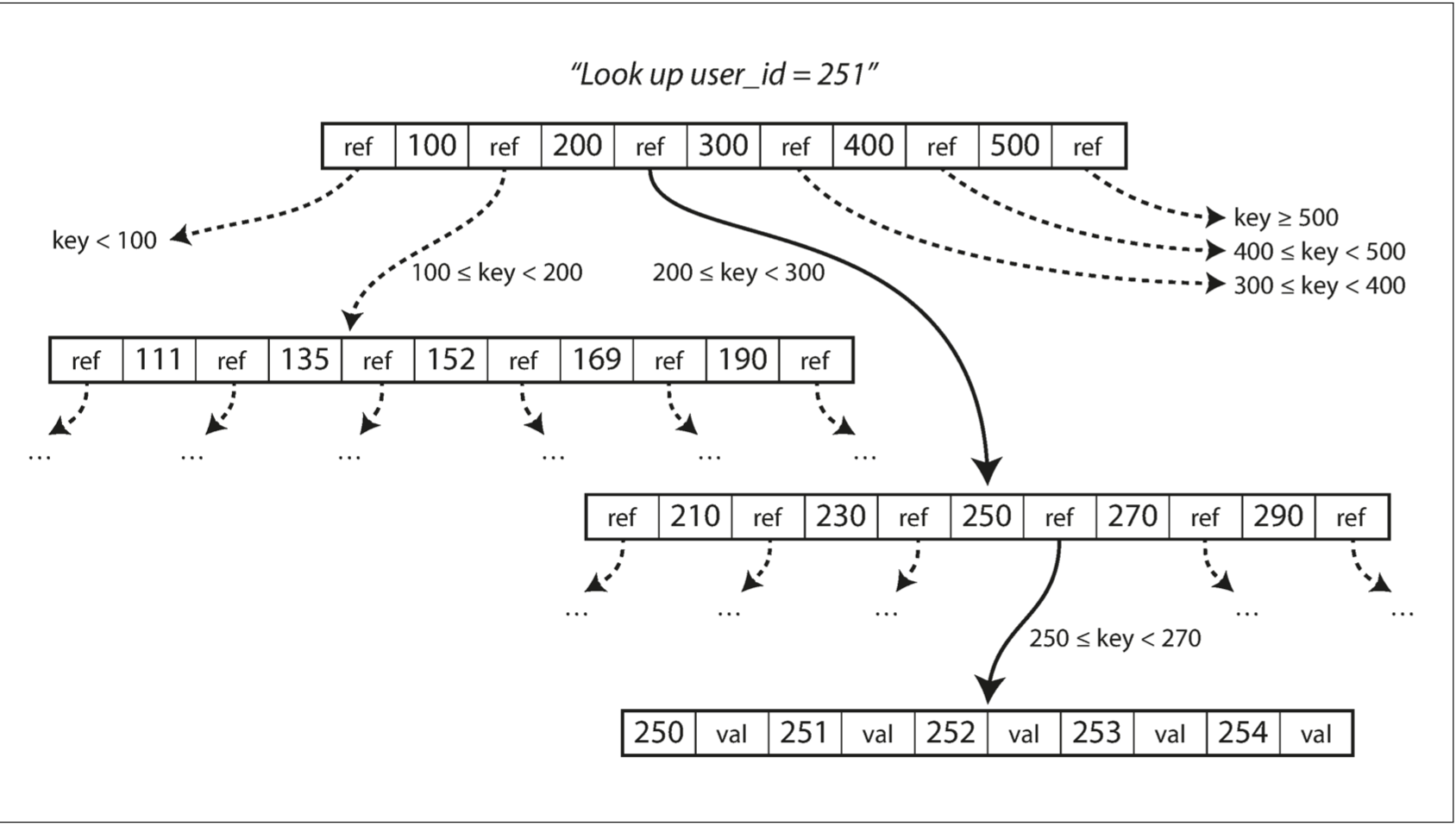
**B-TREES**

B-Trees have stood the test of time very well. They remain the standard index implementation in almost all relational databases and some non-relational DB uses them too.

Like SS-Tables B-Tree keep key-value pair for pairs sorted by key , which allows efficient key-value look ups and range queries. B-Trees break the database down into

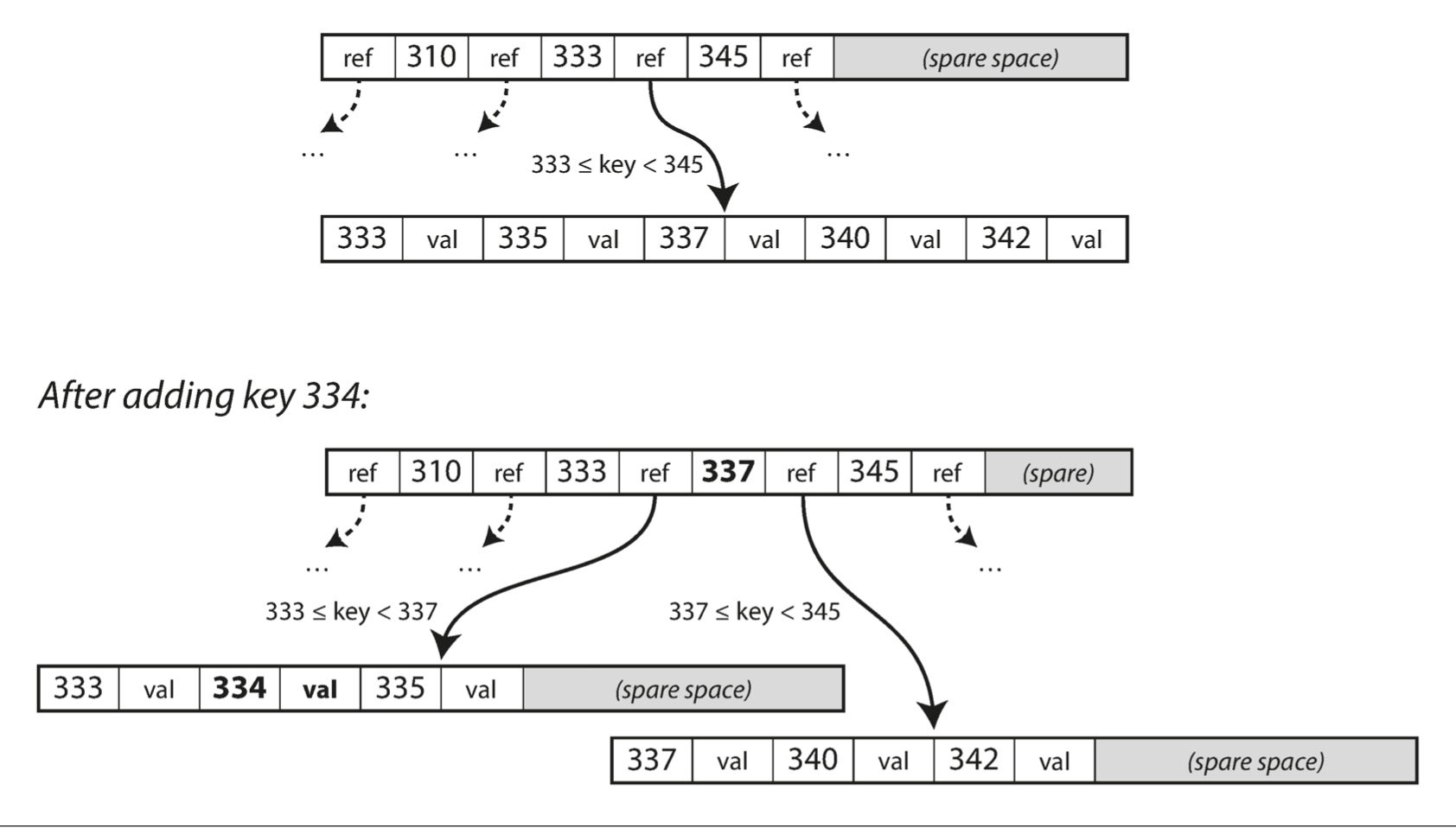
Fixed size blocks or pages, traditionally 4 KB in size (sometimes bigger), and read or write one page at a time.

Each page can be identified using an address, or location, which allows one page to refer to other page, similar to pointer but on disk not memory.



One page is designated as root of the B-Tree, whenever we want to look up a key in index, we start from here, and then we can traverse, there are ranges in each child and we can look for the exact key based on various ranges like example shown in images.

The number of references to child pages in one page is called branching factor. Like in this example its 6, Typically its several hundreds. If we want to add a new key, we have to reach the range where it will lie and then add, but if no space is available then a page will be split into two two half-full pages, and the parent page is updated to account for the new subdi‐ vision of key ranges.



The algorithm ensures that Tree remains balanced : B-Tree with n keys always has O(log n). Most databases can fit into a B-Tree that is three or four levels deep, so we don’t need lots of references to find our page(A four-level tree of 4 KB pages with a branching factor of 500 can store up to 256 TB.) .

**Making B-Tress Resilient**

In order to make database resilient to crashes, it is common for b-trees implementation to include an additional DS called “Write ahead Log” (WAL also known as redo log). This is append only file, where every modification is written before it is applied to pages of the tree itself. When the database comes back after crash, this log is used to restore the tree back to its consistent state.

We also need to maintain concurrency if multiple threads are going to access B-tree at the same time. This is done by protecting tree’s data structure with latches (lightweight locks). Log structured approaches are simple in this regard because they do all the merging in the background with disturbing the new queries and automatically swap new segments from old one time to time.

**B-Tree Optimizations**

Over the period of time there are lots of optimizations are done on B-Trees :

* Instead of overwriting pages and maintaining a WAL for crash recovery, some DB like LMDB use a copy-on-write scheme. A modified page is written to a different location and a new version of parent pages of the tree is created, pointing to the new location.
* We can save space in pages by not storing entire key but abbreviating it, we only need enough info to find out the next child in next level.
* Additional pointers have been added to the tree. For example, each leaf page may have references to its sibling pages to the left and right, which allows scanning keys in order without jumping back to parent pages.
* B-tree variants such as *fractal trees* borrow some log-structured ideas to reduce disk seeks.

Comparison of LSM and B-Trees :

Advantages of LSM Trees :

* Moreover, LSM-trees are typically able to sustain higher write throughput than B- trees, partly because they sometimes have lower write amplification (This effect—one write to the database resulting in multiple writes to the disk over the course of the database’s lifetime ) and partly because they sequentially write compact SSTable files rather than having to overwrite several pages in the tree [26]. This difference is particularly important on magnetic hard drives, where sequential writes are much faster than random writes.
* LSM Trees can be compressed better, and thus often produce smaller files on disk then B-Trees. B-Trees leaves some disk unused due to fragmentation. Periodic compaction and merging done by LSM-Tree help in this to reduce unused space.

Disadvantages of LSM Trees :

* In LSM trees storage compaction can sometime interfere with the performance of read and writes.
* Another issue with compaction arises at high write throughput: the disk’s finite write bandwidth needs to be shared between the initial write (logging and flushing a memtable to disk) and the compaction threads running in the background.
* Sometime compaction is not able to keep up with incoming write operations, and reads become slow due to more lookups in ss tables.
* Duplicate key can exist in LSM and therefore locking mechanism also becomes complex.
* B-trees are very ingrained in the architecture of databases and provide consistently good performance for many workloads, so it’s unlikely that they will go away anytime soon. In new datastores, log-structured indexes are becoming increasingly popular. There is no quick and easy rule for determining which type of storage engine is better for your use case, so it is worth testing empirically.

In a database management system (DBMS), indexes are data structures that improve the speed of data retrieval operations on tables. They work similarly to indexes in a book, allowing you to quickly locate specific information without scanning the entire table. Here's an overview of index types commonly used in databases:

1. **Index:**
   * An index in a database is a data structure that organizes the values of one or more columns in a table to facilitate efficient data retrieval.
   * When you create an index on a table, the DBMS creates a separate data structure that contains pointers to the rows in the table, based on the indexed column(s).
   * Indexes are used to speed up SELECT queries by reducing the number of rows that need to be scanned to find the desired data.
2. **Clustered Index:**
   * A clustered index is an index in which the physical order of rows in the table matches the order of the index.
   * In a clustered index, the rows of the table are stored in the same order as the index key. This means that the data pages of the table are physically ordered based on the clustered index key.
   * Each table can have only one clustered index because the data rows can be ordered in only one way.
   * Clustered indexes are often used on columns that are frequently used in range queries (e.g., date ranges) or to retrieve sequential data efficiently.
3. **Non-Clustered Index:**
   * A non-clustered index is an index in which the physical order of rows in the table does not match the order of the index.
   * In a non-clustered index, the index key values are stored in a separate data structure, and each index entry contains pointers to the corresponding rows in the table.
   * Unlike clustered indexes, a table can have multiple non-clustered indexes because they do not affect the physical order of data in the table.
   * Non-clustered indexes are useful for speeding up searches on columns that are frequently used in WHERE clauses, JOIN conditions, or ORDER BY clauses.

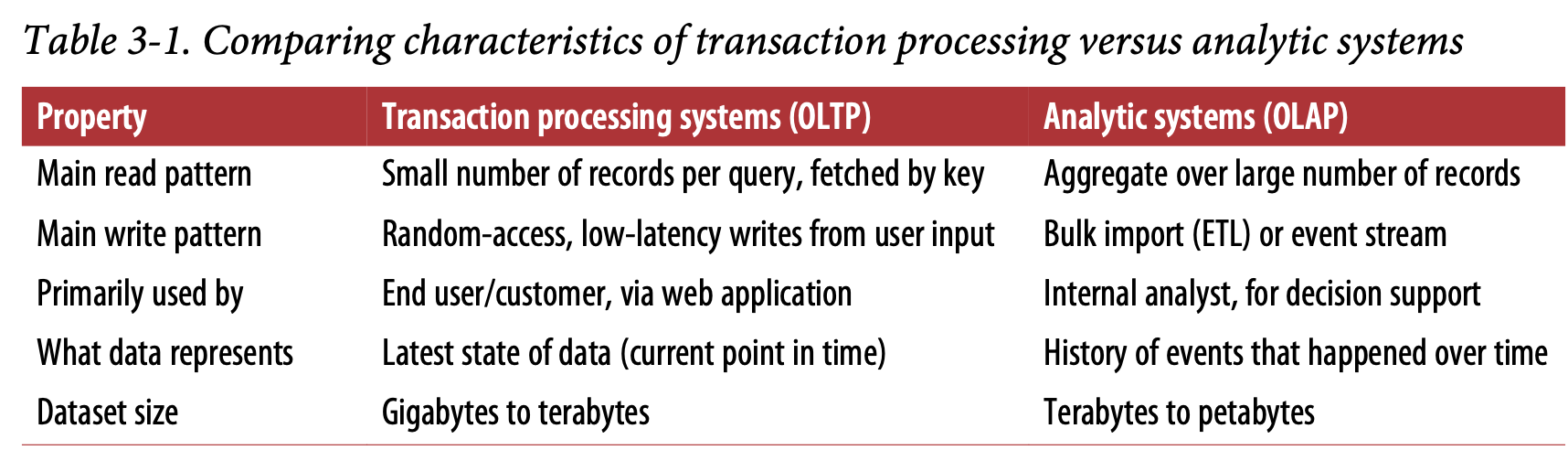
A compromise between a clustered-index (storing all row data within the index) and a non-clustered index (storing only references to the data within the index) is known as covering index or index with included columns, which stores *some* of a table’s columns within the index. This allows some queries to be answered by using the index alone (in which case, the index is said to *cover* the query).

As with any kind of duplication of data, clustered and covering index can speed of reads, but they require additional storage and can add overheads on writes. Databases also need to make additional effort maintain transactional consistency.

TRANSACTION PROCESSING AND ANALYTICS

In early days of business data processing, a write to the DB corresponds to a commercial transaction taking place: making a sale, placing a order with supplier etc. We used database for different kinds of data like comments on blog post, action in game, Ecommerce transaction data etc, but the basic access pattern remains same as similar to processing business transaction. An application typically looks up a small number of records using some key or index, records are inserted or updated on user’s input, because these applications are interactive, the access pattern became knows as ONLINE TRANSACTION PROCESSING (OLTP).

However Databases also started using for Data Analytics, which has very different access pattern. Usually analytics query need to scan over huge number of records, reading few column of a record and calculate aggregate statistic such as SUM, AVERRAGE, MEAN etc, rather than returning raw data to user. For ex : If you are a shoe seller company then statistic might be to find best selling shoe over last six month. These queries are often written by business analysts and results are used to make better business decisions (Business Intelligence). This pattern is known as ONLINE ANALYTIC PROCESSING (OLAP).

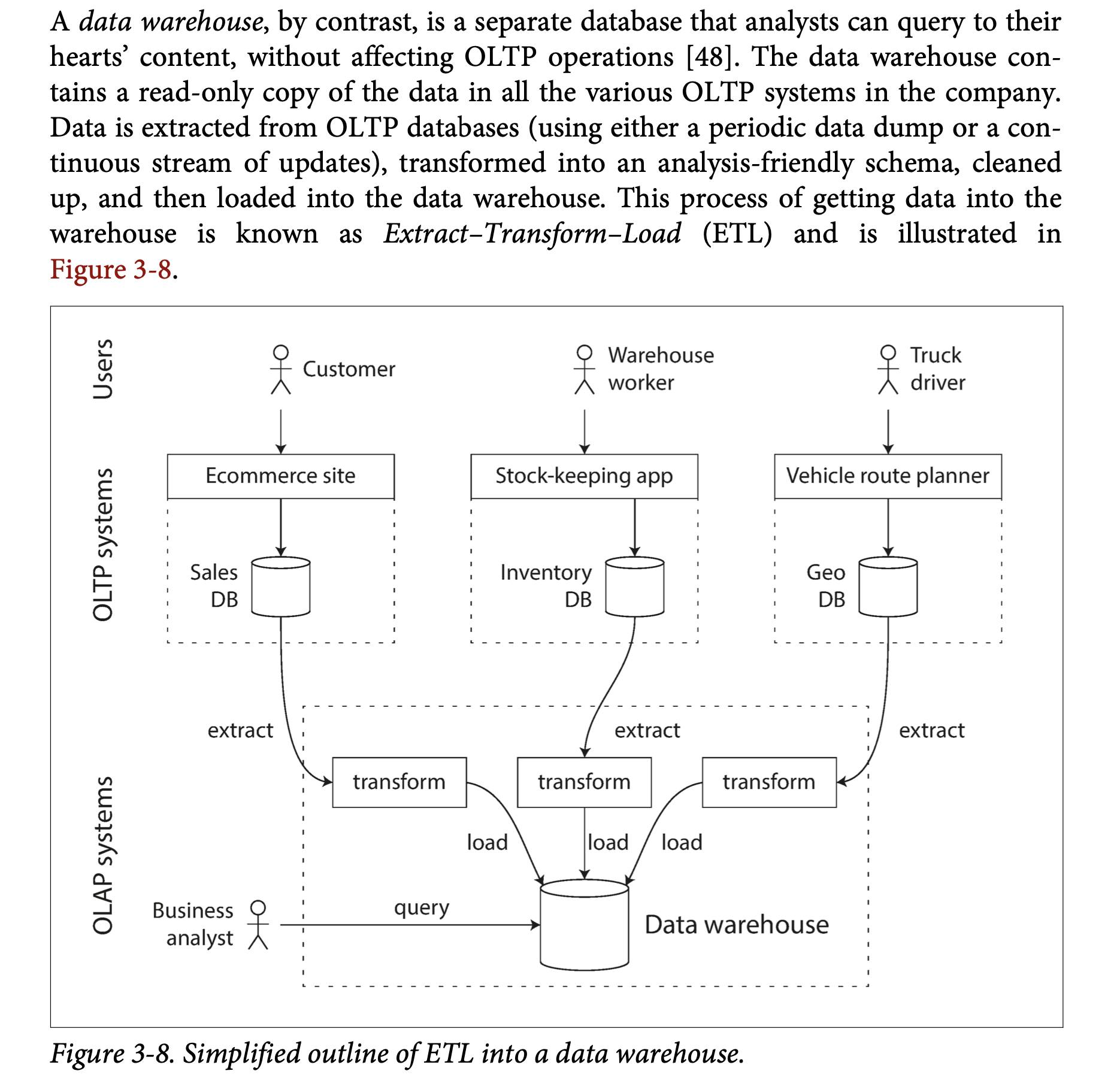


Earlier same databases were used for transaction and analytics both, but now companies use separate DBs to run analytics over them and this separate database is called DATA WAREHOUSE.

DATA WAREHOUSING

An Enterprise might have dozens of different transaction processing system each handling customer requests like tracking inventory, handling order requests and what not and making business work. Each of the system is complex and need people to maintain it, so the system end up operating mostly autonomous from each other. The OLTP systems are usually expected to be highly available and to process transaction with low latency, since they are often critical to the operation of business.

DB administrator closely guard their OLTP databases. And needless to say business analysts run ad-hoc analytic queries on a OLTP databases since those queries are often expensive, scanning large parts of the dataset, which can harm the performance of concurrently executing transactions.



DIVERGENCE BETWEEN OLTP DATABASES AND DATA WAREHOUSES

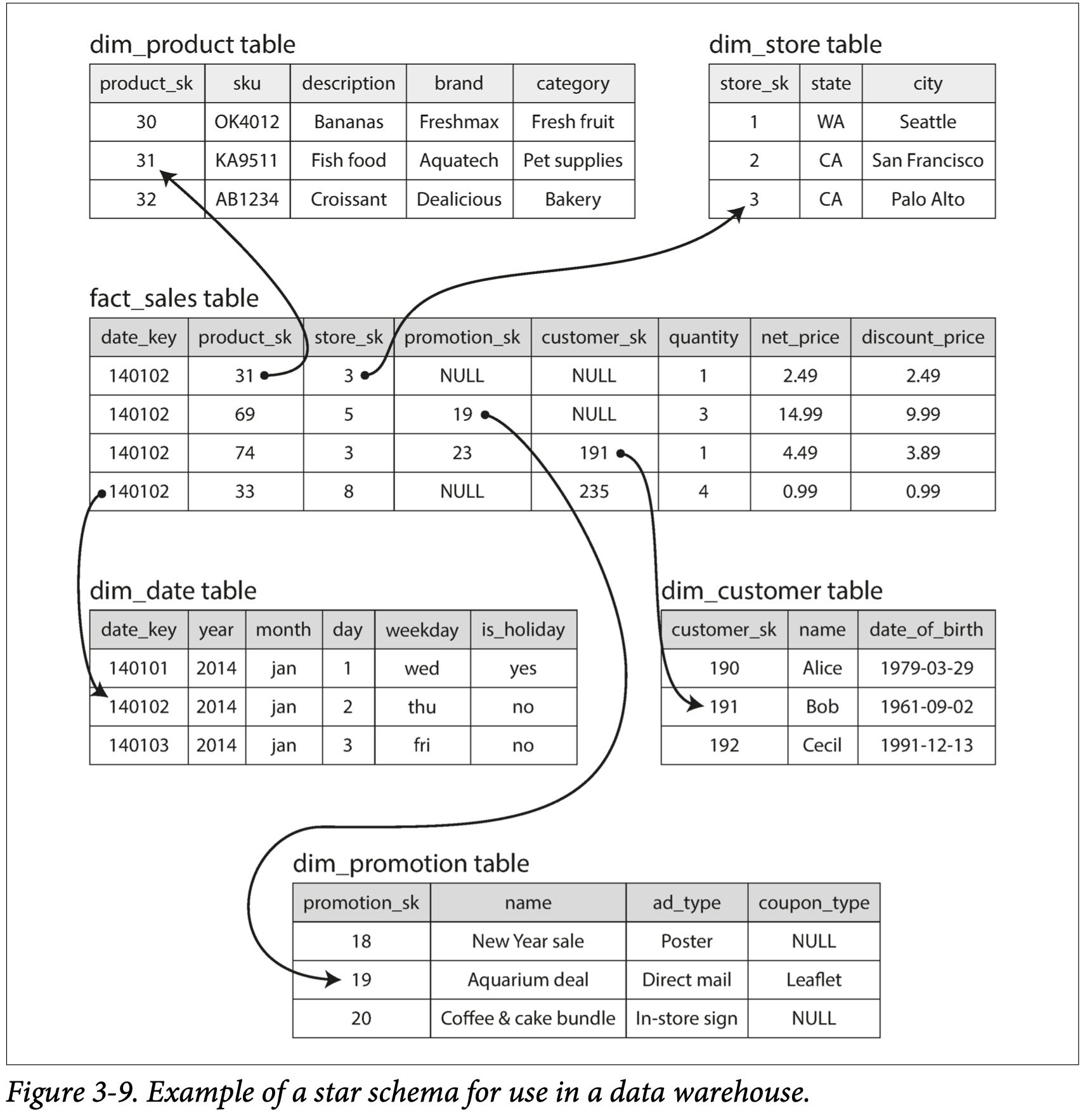
The data model of a data warehouse is generally relational because SQL is generally a good fit for analytics. On interface OLTP and Data warehousing looks similar but internals of system can look quite different and optimized differently for better performance, that’s why most of the DB vendor now support either transactional processing or analytics workload but not both.

Some databases, such as Microsoft SQL Server and SAP HANA, have support for transaction processing and data warehousing in the same product. However, they are increasingly becoming two separate storage and query engines, which happen to be accessible through a common SQL interface

Data warehouse vendors such as Teradata, Vertica, SAP HANA, and ParAccel typi‐ cally sell their systems under expensive commercial licenses. Amazon RedShift is a hosted version of ParAccel. More recently, a plethora of open source SQL-on- Hadoop projects have emerged; they are young but aiming to compete with commercial data warehouse systems. These include Apache Hive, Spark SQL, Cloudera Impala, Facebook Presto, Apache Tajo, and Apache Drill [52, 53]. Some of them are based on ideas from Google’s Dremel [54].

STARS AND SNOWFLAKES: Schema For Analytics

Unlike Transactional processing, in analytics there is much less diversity of data models. Many data warehouses are used in formulatic style known as STAR SCHEMA (also known as dimensional modelling).



The name “star schema” comes from the fact that when the table relationships are visualized, the fact table is in the middle, surrounded by its dimension tables; the connections to these tables are like the rays of a star.

The name “star schema” comes from the fact that when the table relationships are visualized, the fact table is in the middle, surrounded by its dimension tables; the connections to these tables are like the rays of a star.

A variation of this template is known as the *snowflake schema*, where dimensions are further broken down into subdimensions. For example, there could be separate tables for brands and product categories, and each row in the dim\_product table could ref‐ erence the brand and category as foreign keys, rather than storing them as strings in the dim\_product table. Snowflake schemas are more normalized than star schemas, but star schemas are often preferred because they are simpler for analysts to work with [55].

In a typical data warehouse, tables are often very wide: fact tables often have over 100 columns, sometimes several hundred [51]. Dimension tables can also be very wide, as they include all the metadata that may be relevant for analysis—for example, the dim\_store table may include details of which services are offered at each store, whether it has an in-store bakery, the square footage, the date when the store was first opened, when it was last remodeled, how far it is from the nearest highway, etc.

COLUMN - ORIENTED STORAGE

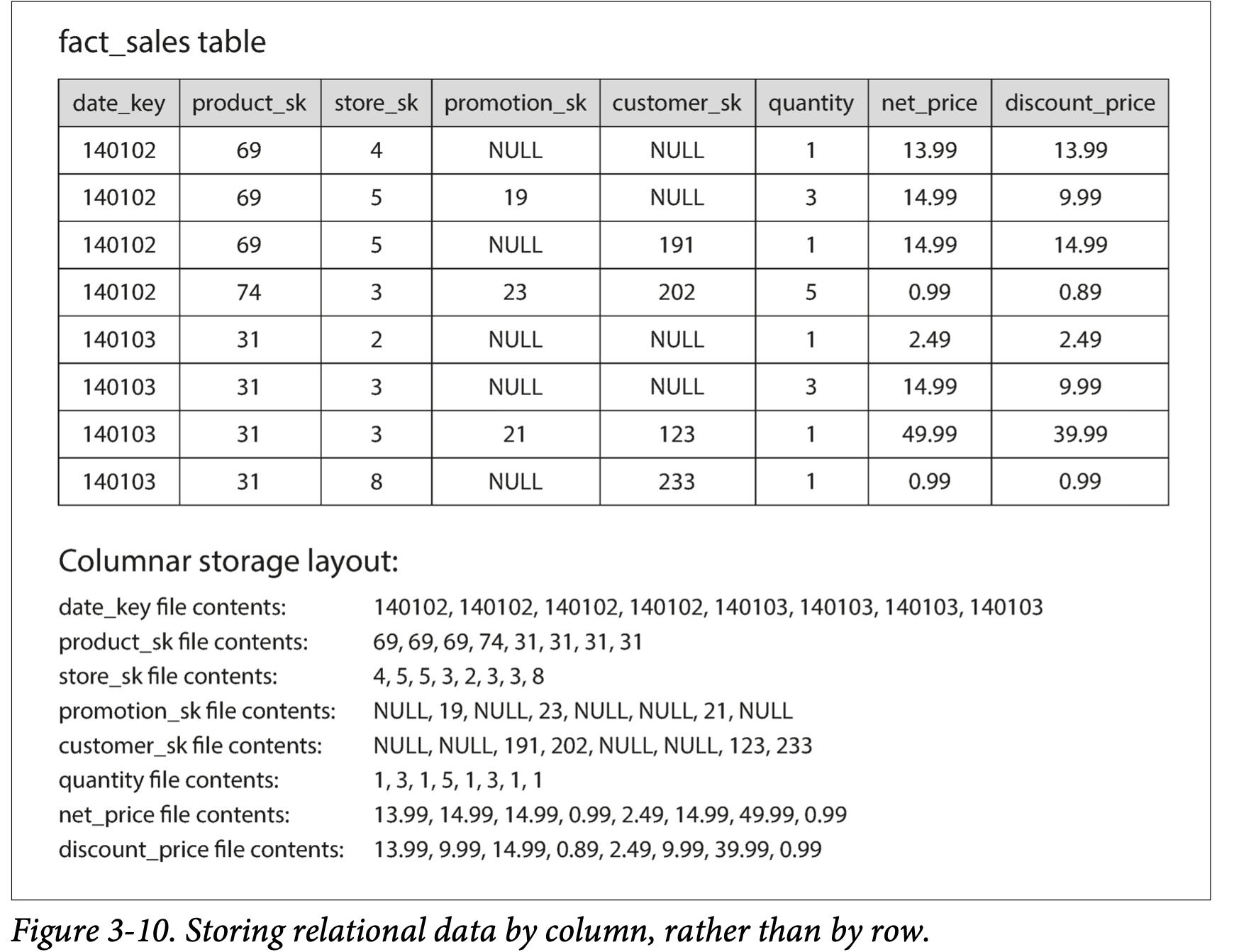
If we have trillions of rows and petabyte of data in our fact table, then storin and efficiently querying them becomes a huge problem. Although fact table may be over 100 column wide but at time data warehouse query only accesses 4-5.

Ex: A computer code with text

Description automatically generated

In most OLTP databases storage is laid out in row-oriented fashion, all the values from one row of a table are stored next to each other. You may have indexes on fact\_sales.date\_key and/or fact\_sales.product\_sk that tell the storage engine where to find all the sales for a particular date or for a particular product. But then, a row-oriented storage engine still needs to load all of those rows (each consisting of over 100 attributes) from disk into memory, parse them, and filter out those that don’t meet the required conditions. That can take a long time.

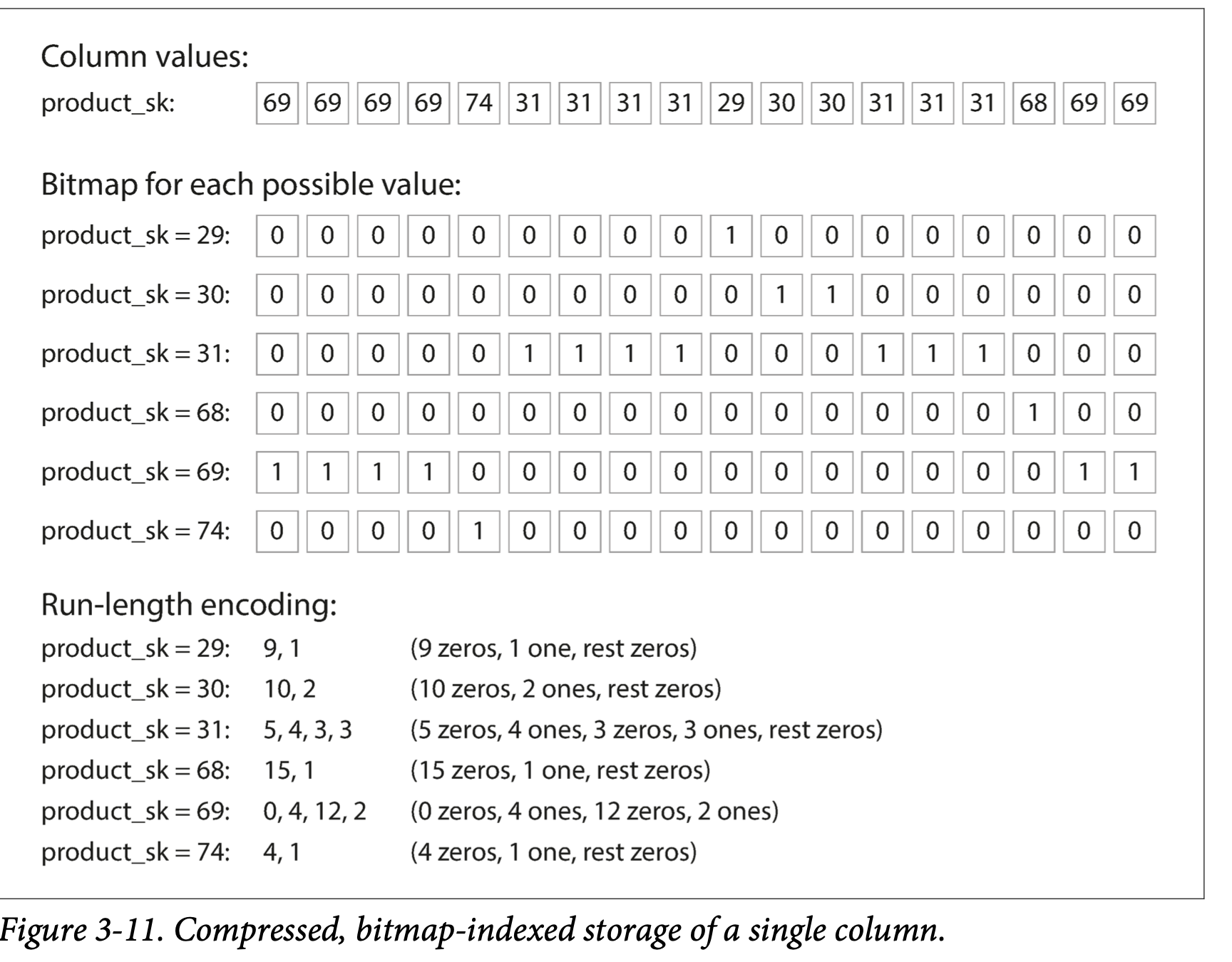
We can store data in column-oriented fashion. The idea is to store all the values from one column together into a file, so whichever column is involved in the query can be used separately instead of loading all data.



And each column file will contain row in same order so if need be to reassemble the complete row you can do that easily by picking simultaneously taking nth row value from all column files.

COLUMN COMPRESSION

We can further reduce demand on disk by using column compression technique, One technique that is particu‐ larly effective in data warehouses is BITMAP ENCODING.

**

Besides reducing the volume of data that needs to be loaded from disk, column- oriented storage layouts are also good for making efficient use of CPU cycles. For example, the query engine can take a chunk of compressed column data that fits comfortably in the CPU’s L1 cache and iterate through it in a tight loop (that is, with no function calls). A CPU can execute such a loop much faster than code that requires a lot of function calls and conditions for each record that is processed. Column compression allows more rows from a column to fit in the same amount of L1 cache. Operators, such as the bitwise *AND* and *OR* described previously, can be designed to operate on such chunks of compressed column data directly. This technique is known as *vectorized processing.*

SORT ORDER IN COLUMN STORAGE

We can sort rows in according to one column, based on what types of queries are coming, for example if querier are coming heavily based on date ranges then it might be a good idea to sort them according to date ranges, then we can implement indexing on them. Similarly we can include second or more sort keys to refine and make our searches more faster.

SEVERAL DIFFERENT SORT ORDERS

A clever extension of this idea was introduced in C-Store and adopted in the commercial data warehouse Vertica. We often store redundant data on multiple machines so that, if one machine loses its data it can be recovered from other one. So why not store replicated data on different machines based on different sort column, so that query can be efficiently answered based on best sort key available on any machine. Its similar to having multiple secondary indexing.

WRITING TO COLUMN - ORIENTED STORAGE

These optimizations are done for data warehouses where all these things like making column-oriented storage and compressing the data makes queries reads faster, but at the same time make writes difficult. An update in-place approach like B-Trees use won’t work here, If you want to insert a row in the middle, most likely you have to rewrite all the column files As rows are identified by their position within a column, the insertion has to update all columns consistently.

Fortunately, we have already seen a good solution earlier in this chapter: LSM-trees. All writes first go to an in-memory store, where they are added to a sorted structure and prepared for writing to disk. It doesn’t matter whether the in-memory store is row-oriented or column-oriented. When enough writes have accumulated, they are merged with the column files on disk and written to new files in bulk. This is essen‐ tially what Vertica does.

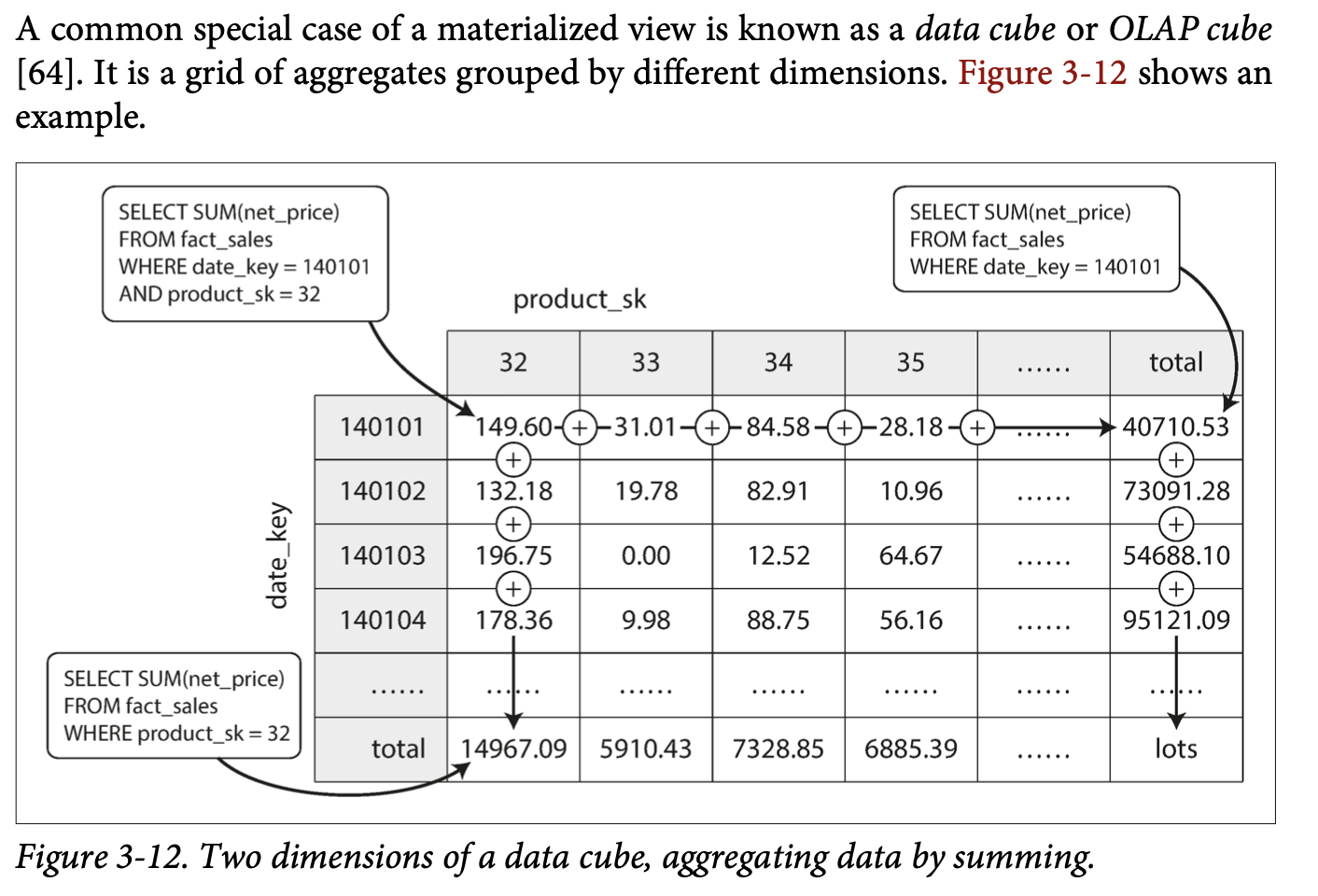
Queries need to examine both the column data on disk and the recent writes in mem‐ ory, and combine the two. However, the query optimizer hides this distinction from the user. From an analyst’s point of view, data that has been modified with inserts, updates, or deletes is immediately reflected in subsequent queries.

AGGREGATION : DATA CUBES AND MATERIALIZED VIEWS

Another important aspect of data warehouses is materialized aggregates, data warehouse queries are often resolved around aggregate functions such as COUNT, SUM, AVG, MIN, MAX in SQL. If same aggregate is used my many queries why not cache it.

One way of creating such as view is materialized view. In a relational data model, it is often defined like standard (virtual) view : A table like object whose content is the result of some query. The difference is that a materialized view is an actual copy of the query results, written to disk, whereas a virtual view is just a shortcut for writing queries. When you read from a virtual view, the SQL engine looks into the view’s underlying query on the fly and then processes the expanded query (actual query).

When the underlying data changes, a materialized view needs to be updated, because it is a denormalized copy of the data. The database can do that automatically, but such updates make writes more expensive, which is why materialized views are not often used in OLTP databases.



MODES OF DATAFLOW

We discussed whenever we want to send some data to some another process with which we don’t share memories, for ex : sending data over network or writing to a file, we need to encode data as sequence of bytes. And we discussed variety of encoding for this.

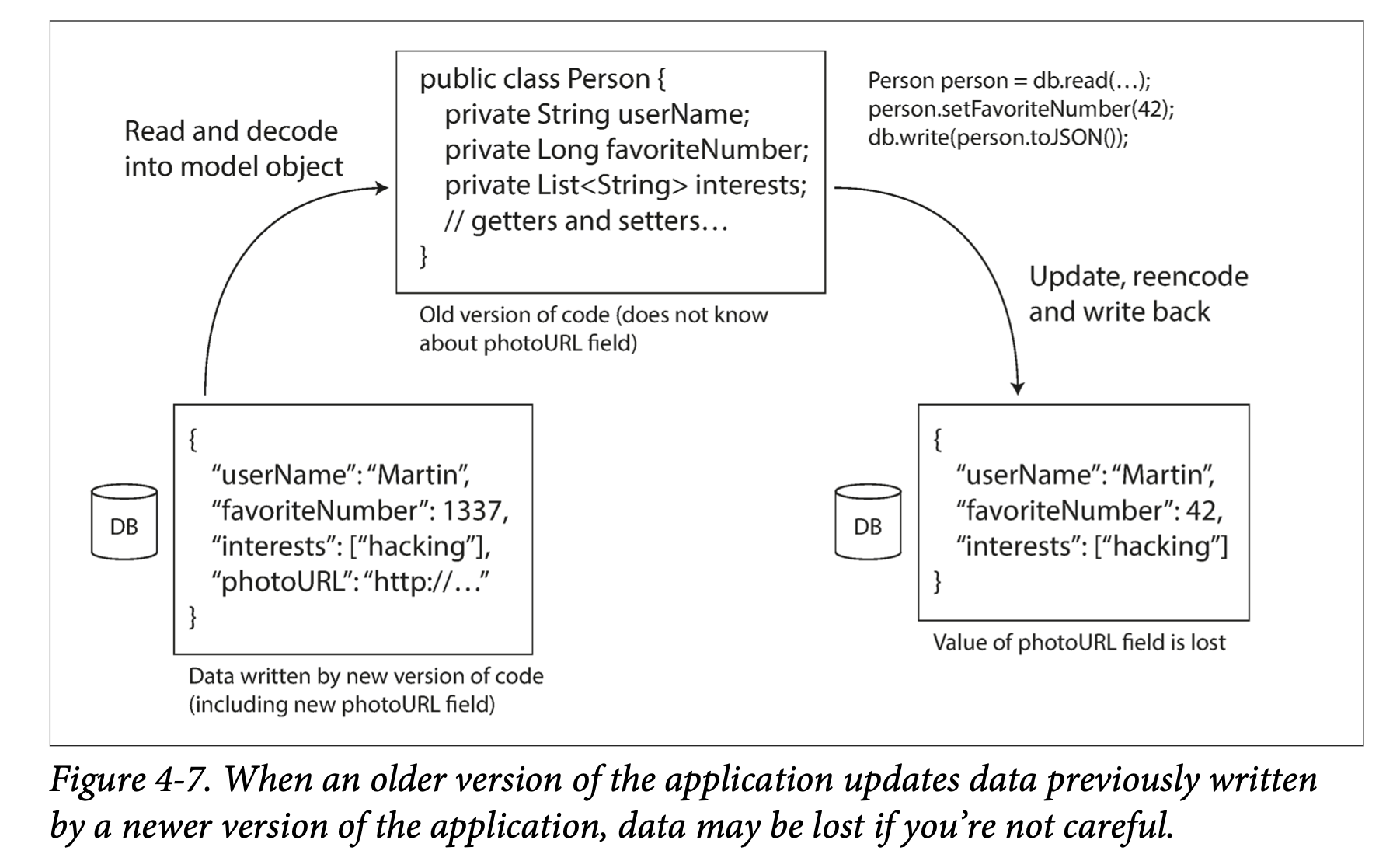
We talked about forward and backward compatibility, which are important for evolvability (making change easy by allowing you to upgrade different parts of your system independently, and not having to change everything at once). Compatibility is a relationship between one process that encodes the data, and another process that decodes it. Now there are many ways data can flow from one process to another :

DATAFLOW THROUGH DATABASES

In a database, process that writes to database encodes it and process that reads form it decodes it. There may be a single process accessing the database, in which case the reader is just simply a later version of the same process, in that case we can think of like that we are sending message to future self.

Backward compatibility is absolutely necessary here otherwise future self won’t be able to decode what we previously wrote. It’s very common that multiple process accesses a database at the same time. In a environment where application is changing, it is likely that some processes accessing the database will be running newer code and some will be running older code – for ex : when a new version is currently deployed in a rolling upgrade, so some instances have been updated which some might not.

This means value in DB might be written by newer code and read by older code so, forward compatibility also required for databases. Sometimes we need to address these things at application level.



There are some other issue that can come, for ex : DB allows writing of data at any time, there may be data present from 5 years ago and some new data that is just millisecond old, but problem lies when different encoding is used for storage at different times. This observation is sometimes summed up as data outlives code. Rewriting (*migrating*) data into a new schema is certainly possible, but it’s an expensive thing to do on a large dataset, so most databases avoid it if possible. Most relational databases allow simple schema changes, such as adding a new column with a null default value, without rewriting existing data.v When an old row is read, the database fills in nulls for any columns that are missing from the encoded data on disk. LinkedIn’s document database Espresso uses Avro for storage, allowing it to use Avro’s schema evolution rules.

DATAFLOW THROUGH REST AND RPC

Skipped content about REST and SOAP…..

THE PROBLEMS WITH REMOTE PROCEDURE CALLS (RPCs)

Web services like REST are latest incarnation of a long line of technologies for making APIs request over a network, but had many problems. Enterprise JavaBeans (EJB) and Java’s Remote Method Invocation (RMI) are limited to Java. The Distributed Component Object Model (DCOM) is limited to Microsoft platforms. The Common Object Request Broker Architecture (CORBA) is excessively complex, and does not provide backward or forward compatibility. All of these are based on RPC (Remote Procedure calls) which has been around since 1970s.

The RPC model tries to make a request to a remote network service looks same as calling function or method in programming language, within the same process (this abstraction is called as location transparency). RPC has varity of problems within itself and differs in various ways from a function or method calls :

1. A local function call is completely in your control, depending on params it passes, fails or returns something. But a network call may fail due to variety of reasons like unavailability of servers, slow etc. Thus you have to anticipate them. Similarly a function may return a result, throw exception or never returns maybe due to infinite loop, but a network request has another outcome, it may return a result without a outcome, due to timeout, and you never know the reasons of that.
2. It may happen that request is getting through only response is lost, in that case there will be duplicate operations being performed , unless you build a mechanism for deduplication (*idempotence*) into the protocol.
3. Network call latency is widely variable unlike function calls which takes almost same time for every call.
4. Passing parameter and references is easy in method calling but in network call you have to encode and it becomes complex for big objects.
5. The client and the service may be implemented in different programming languages, so the RPC framework must translate datatypes from one language into another. This can end up ugly, since not all languages have the same types

MESSAGE PASSING DATAFLOW

we will briefly look at *asynchronous message-passing* systems, which are somewhere between RPC and databases. They are similar to RPC in that a client’s request (usually called a *message*) is delivered to another process with low latency. They are similar to databases in that the message is not sent via a direct net‐ work connection, but goes via an intermediary called a *message broker* (also called a *message queue* or *message-oriented middleware*), which stores the message temporarily.

Using a message broker has several advantages compared to direct RPC:

* It can act as a buffer if the recipient is unavailable or overloaded, and thus improve system reliability.
* It can automatically redeliver messages to a process that has crashed, and thus prevent messages from being lost.
* It avoids the sender needing to know the IP address and port number of the recipient (which is particularly useful in a cloud deployment where virtual machines often come and go).
* It allows one message to be sent to several recipients.
* It logically decouples the sender from the recipient (the sender just publishes messages and doesn’t care who consumes them).

However, a difference compared to RPC is that message-passing communication is usually one-way: a sender normally doesn’t expect to receive a reply to its messages. It is possible for a process to send a response, but this would usually be done on a separate channel. This communication pattern is *asynchronous*: the sender doesn’t wait for the message to be delivered, but simply sends it and then forgets about it.