Databases 2022

Darko Bozhinoski,
Ph.D. in Computer Science

Agenda

- Database concurrency control: Two-Phase Locking Techniques
- Storage Architecture
- NoSQL Databases

Lifecycle of transaction execution

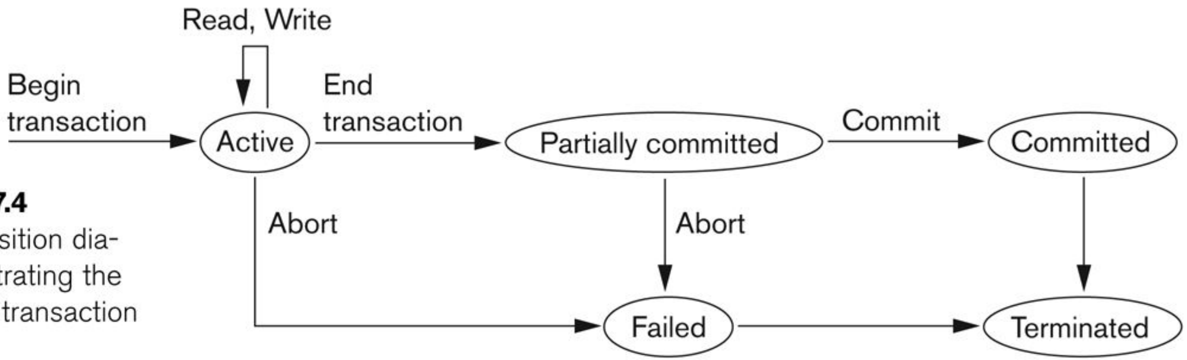


Figure 17.4
State transition diagram illustrating the states for transaction execution.

Serializable schedule

- **Serializable schedule:**
  - A schedule S is serializable if it is equivalent to some serial schedule of the same n transactions.
  - Resolves the issue with serial schedule: they limit concurrency by prohibiting interleaving of operations.

Practical approach to creating serializable schedules

- Protocols, or rules, are developed that guarantee that any schedule that follows these rules will be serializable.
- It’s not possible to determine when a schedule begins and when it ends.
  - We reduce the problem of checking the whole schedule to checking only a **committed projection** of the schedule (i.e. operations from only the committed transactions.)
- Current approach used in most DBMSs:
  - Use of locks with **two phase locking**

Database Concurrency Control

- **Purpose of Concurrency Control**
  - To enforce Isolation (through mutual exclusion) among conflicting transactions.
  - To preserve database consistency through consistency preserving execution of transactions.
  - To resolve read-write and write-write conflicts.

- Example:
  - In concurrent execution environment if T1 conflicts with T2 over a data item A, then the existing concurrency control decides if T1 or T2 should get the A and if the other transaction is rolled-back or waits.

Binary locks for Concurrency Control

- Locking is an operation which secures
  - (a) permission to Read
  - (b) permission to Write a data item for a transaction.
- Example:
  - Lock (X). Data item X is locked in behalf of the requesting transaction.
- Unlocking is an operation which removes these permissions from the data item.
- Example:
  - Unlock (X): Data item X is made available to all other transactions.
- Lock and Unlock are Atomic operations.

Binary locks for Concurrency Control: Essential components

- **Two locks modes:**
  - (a) shared (read) (b) exclusive (write).
- **Shared mode: shared lock (X)**
  - More than one transaction can apply share lock on X for reading its value but no write lock can be applied on X by any other transaction.
- **Exclusive mode: Write lock (X)**
  - Only one write lock on X can exist at any time and no shared lock can be applied by any other transaction on X.

Binary locks for Concurrency Control: Essential components

- **Lock Manager:**
  - Managing locks on data items.
- **Lock table:**
  - Lock manager uses it to store the identify of transaction locking a data item, the data item, lock mode and pointer to the next data item locked. One simple way to implement a lock table is through linked list.

Transaction ID	Data item id	lock mode	Ptr to next data item
T1	X1	Read	Next

Binary locks for Concurrency Control

- **Database requires that all transactions should be well-formed. A transaction is well-formed if:**
  - It must lock the data item before it reads or writes to it.
  - It must not lock an already locked data items and it must not try to unlock a free data item.

Lock Operation

```
B: if LOCK (X) = 0 (*item is unlocked*)
   then LOCK (X) ← 1 (*lock the item*)
   else begin
     wait (until lock (X) = 0) and
     the lock manager wakes up the transaction);
   goto B
   end;
```

Unlock Operation

*LOCK (X) ← 0 (*unlock the item*)*

if any transactions are waiting then

wake up one of the waiting the transactions;

Serializability and Two-phase Locking

<u>Binary locks or read/write locks do not guarantee serializability of schedule on its own!</u>

The items Y in T1 and X in T2 were unlocked too early. This allows a schedule such as the one in c) to occur which is not a serializable schedule -> incorrect schedule.

To guarantee serializability, we must follow two-phase locking protocol.

(a)

T_1	T_2

read_lock(Y);
read_item(Y);
unlock(Y);
write_lock(X);
read_item(X);
$X := X + Y$;
write_item(X);
unlock(X);
	read_lock(X);
	read_item(X);
	unlock(X);
	write_lock(Y);
	read_item(Y);
	$Y := X + Y$;
	write_item(Y);
	unlock(Y);

(b) Initial values: X=20, Y=30

Result serial schedule T_1 followed by T_2: X=50, Y=80

Result of serial schedule T_2 followed by T_1: X=70, Y=50

(c)

Time

T_1	T_2

read_lock(Y);
read_item(Y);
unlock(Y);	

	read_lock(X);
	read_item(X);
	unlock(X);
	write_lock(Y);
	read_item(Y);
	$Y := X + Y$;
	write_item(Y);
	unlock(Y);

write_lock(X);
read_item(X);
$X := X + Y$;
write_item(X);
unlock(X);	

Result of schedule S:
X=50, Y=50
(nonserializable)

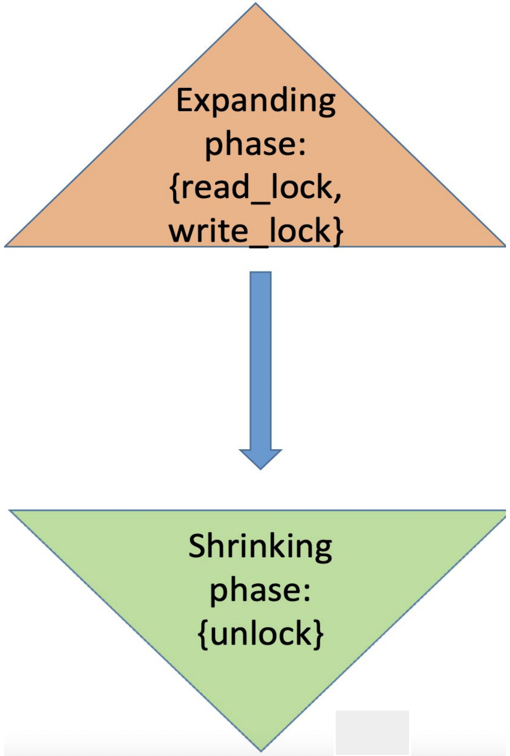
Figure 21.3
Transactions that do not obey two-phase locking. (a) Two transactions T_1 and T_2. (b) Results of possible serial schedules of T_1 and T_2. (c) A nonserializable schedule S that uses locks.

Two-phase Locking (2PL)

A transaction follows the basic two-phase locking protocol if all locking operations (read_lock, write_lock) precede the first unlock operation in the transaction.

Two phases:

- An expanding phase, during which new locks on items can be acquired but none can be released;
- A shrinking phase during which existing locks can be released but no new locks can be acquired.



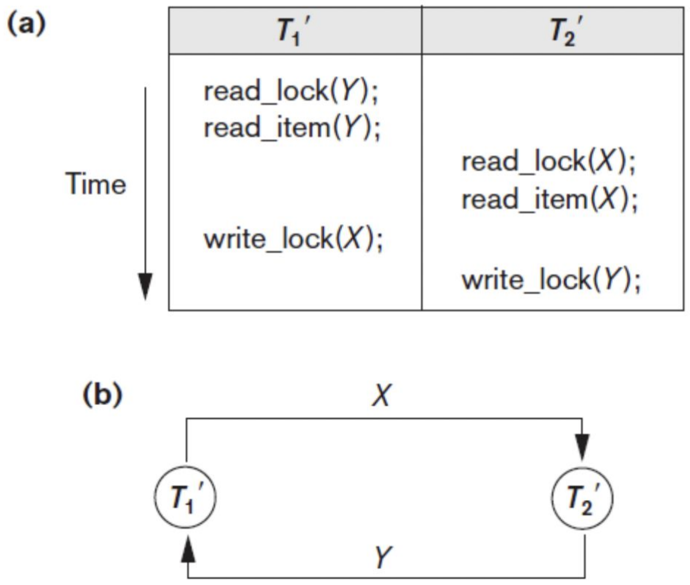
Two-phase policy locking algorithms

- Two-phase policy generates two locking algorithms
  - (a) **Basic**
  - (b) **Conservative**
- **Conservative**:
  - Prevents deadlock by locking all desired data items before transaction begins execution.
- **Basic**:
  - Transaction locks data items incrementally. This may cause deadlock which is dealt with.
- **Strict**:
  - A more stricter version of Basic algorithm where unlocking is performed after a transaction terminates (commits or aborts and rolled-back). This is the most commonly used two-phase locking algorithm.

Locks can cause Deadlock and Starvation

Deadlock occurs when each transaction T in a set of two or more transactions is waiting for some item that is locked by other transaction T’ in the set.

Starvation occurs when a transaction cannot proceed for an indefinite period of time while other transactions in the system continue normally. This occurs if the waiting scheme for locked items is unfair.



Dealing with Deadlock and Starvation

- **Deadlock prevention**
  - A transaction locks all data items it refers to before it begins execution.
  - This way of locking prevents deadlock since a transaction never waits for a data item.
  - The conservative two-phase locking uses this approach.

Dealing with Deadlock and Starvation (2)

- **Deadlock detection and resolution**
  - Here, deadlocks are allowed to happen. The scheduler maintains a **wait-for-graph** for detecting cycle. If a cycle exists, then one transaction involved in the cycle is selected (victim) and rolled-back.
  - A wait-for-graph is created using the lock table. As soon as a transaction is blocked, it is added to the graph. When a chain like: Ti waits for Tj waits for Tk waits for Ti or Tj occurs, then this creates a cycle.

Dealing with Deadlock and Starvation (3)

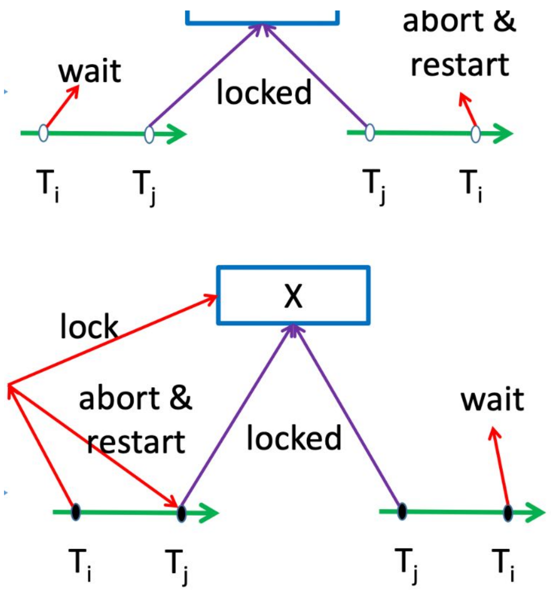
- **Deadlock avoidance**
  - There are many variations of two-phase locking algorithm.
  - Some avoid deadlock by not letting the cycle to complete.
  - That is as soon as the algorithm discovers that blocking a transaction is likely to create a cycle, it rolls back the transaction.
  - Wound-Wait and Wait-Die algorithms use timestamps to avoid deadlocks by rolling-back victim.

Wound-Wait and Wait-Die algorithms

The timestamps are based on the order in which transactions are started. If transaction T1 starts before T2, then TS(T1) ≤ TS(T2). Suppose that Ti tries to lock an item X but is not able to because is locked by Tj with a conflicting lock. The rules are:

<u>Wait-die:</u> an older transaction is allowed to wait for a younger transaction, whereas a younger transaction requesting an item held by an older transaction is aborted and restarted.

<u>Wound-wait:</u> A younger transaction is allowed to wait for an older one, whereas an older transaction requesting an item held by a younger transaction preempts the younger transaction by aborting it.



Dealing with Deadlock and Starvation (4)

- **Starvation**
  - Starvation occurs when a particular transaction consistently waits or restarted and never gets a chance to proceed further.
  - In a deadlock resolution it is possible that the same transaction may consistently be selected as victim and rolled-back.
  - This limitation is inherent in all priority based scheduling mechanisms.
  - In Wound-Wait scheme a younger transaction may always be wounded (aborted) by a long running older transaction which may create starvation.

Storage Architectures

Introduction

The collection of data that makes up a database must be stored physically on some computer storage medium

DBMS can retrieve, update and process data as needed

Locating data on disk a major <u>bottleneck</u> in database applications

The file structures attempt to <u>minimize the number of block transfers</u> needed to locate and transfer data from disk to main memory.

File organization and Access Method

A **file organization** refers to the organization of the data of a file into records, blocks and access structures - including the way records and blocks are placed on the storage medium and interlinked

- Some files may be static, meaning that update operations are rarely performed.
- More dynamic files may change frequently, so updates are constantly applied to them
- Different methods to organize records of a file on disk

An **access method** provides a group of operations that can be applied to a file. It is possible to apply several access methods to a file organized using a certain organization.

- Some access methods can be applied only to files organized in certain ways.

File Organizations

Primary file organizations determine how the file records are physically placed on disk and how the records are accessed.

- A heap file (unaorded file) places records on disk in no particular order by appending new records at the end of the file
- Sorted file (sequential file) keeps the records ordered by the value of a particular field (called sort key)
- Hashed file uses a hash function applied to a particular field to determine a record's placement on disk

A secondary organization or auxiliary access structure allows efficient access to file records based on alternate fields than those that have been used for the primary file organization

- Examples?

File Organizations

Primary file organizations determine how the file records are physically placed on disk and how the records are accessed.

- A heap file (unaorded file) places records on disk in no particular order by appending new records at the end of the file
- Sorted file (sequential file) keeps the records ordered by the value of a particular field (called sort key)
- Hashed file uses a hash function applied to a particular field to determine a record's placement on disk

A secondary organization or auxiliary access structure allows efficient access to file records based on alternate fields than those that have been used for the primary file organization

- Indexes…

Files of Unordered Records

Simplest and most basic type of organization

This organization is often used with additional access paths, such as secondary indexes.

Inserting a new record is very efficient

- The last disk block of the file is copied into a buffer
- The new record is added
- The block is rewritten to disk

To delete a record, a program must first find the block, copy the content to a buffer, delete the record from the buffer and finally rewrite back to the disk.

- It leaves unused space in the disk block
- Requires a periodic reorganization

Searching for a record using any search condition involves a linear search

Files or Ordered Records (Sorted Files)

Records of a file on a disk are ordered based on the values of one of their fields

If the ordering field is also a key field of the file - it is called ordering key for the file

Advantages:

- Reading the records in order of the ordering key values is efficient
- Finding the next record from the current one in order of the ordering key requires no additional block accesses
- Using a search condition based on the value of an ordering key field results in faster access when the binary search technique is used

Disadvantages:

- Inserting and deleting records are expensive operations for an ordered file because the records must remain physically ordered

Hash Files

Characteristics:

- Fast access to records - average search time = $O(1)$
- The search condition must be an equality condition on a single field, called the hash field
- The idea behind hashing is to provide a function h, called a hash function which is applied to the hash field value of a record and yields the address of the disk block in which the record is stored.
- A search for the record within the block can be carried out in a main memory buffer
- For most records, we need only a single block access to retrieve the record

Hash Files (2)

GOALS:

1. <u>To distribute the records uniformly over the address space to minimize collisions.</u>

A collision occurs when the hash field value of a record that is being inserted hasehs to an address that already contains a different record. There are numerous methods for collision resolution that will not be discussed here.

2. <u>To achieve (1) yet occupy the buckets fully, thus not leaving many unused locations.</u> Bucket is either one disk block or a cluster of contiguous disk blocks.

Simulation and analysis studies have shown that it is usually best to keep a hash file between 70% and 90% full so that the number of collisions remains low and we do not waste too much space.

Rules on Making Data Access More Efficient on Disk

1. <u>Buffering of data in memory</u>
2. <u>Proper organization of data on disk</u>: keep related data on contiguous blocks
3. <u>Reading data ahead of request </u>to minimize seek times
4. <u>Proper scheduling of I/O requests</u>
5. <u>Use of log disks</u> to temporarily hold writes
6. <u>Use of SSD f</u>or recovery purposes

NOSQL Databases

Impedance Mismatch

- We need to translate between the relational structure and the organizational needs
  - The **object–relational impedance mismatch** is a set of conceptual and technical difficulties that are often encountered when a relational database management system (RDBMS) is being served by an application program (or multiple application programs) written in an object-oriented programming language or style, particularly because objects or class definitions must be mapped to database tables defined by a relational schema.
- Think about the reports needed for the warehouse
  - Purchase orders
  - History of orders for customer
  - Parts inventory per warehouse
- This means we will need to use Joins
  - This isn’t too much of an issue until we scale …

Speaking of Scaling …

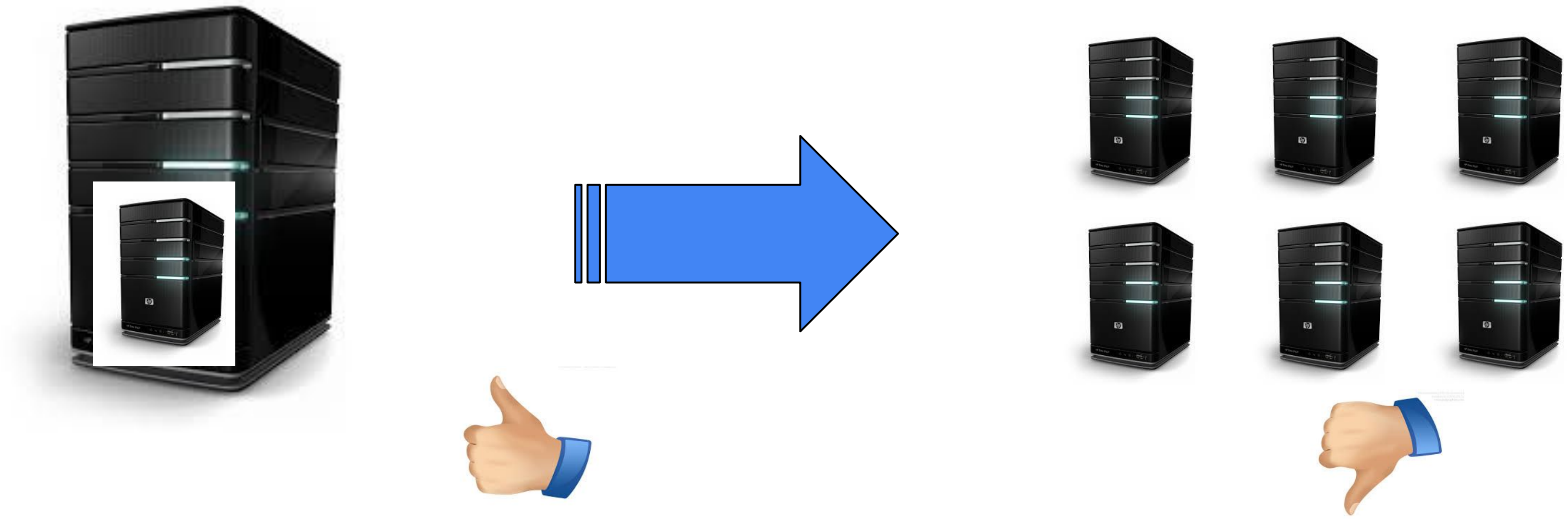
<u>Do relational databases scale?</u>

Speaking of Scaling …

Do relational databases scale?

Vertical scaling - YES;

Horizontal scaling - NO;



Relation Databases Scale Is Difficult

- We can “shard” the data
  - Split the data across the machines
- This is very difficult to do efficiently
- This makes joins more costly
  - Remember joins are common
- This also has a practical limit
  - At some point you will need to replicate the data
- The database becomes slow …

Change is Needed

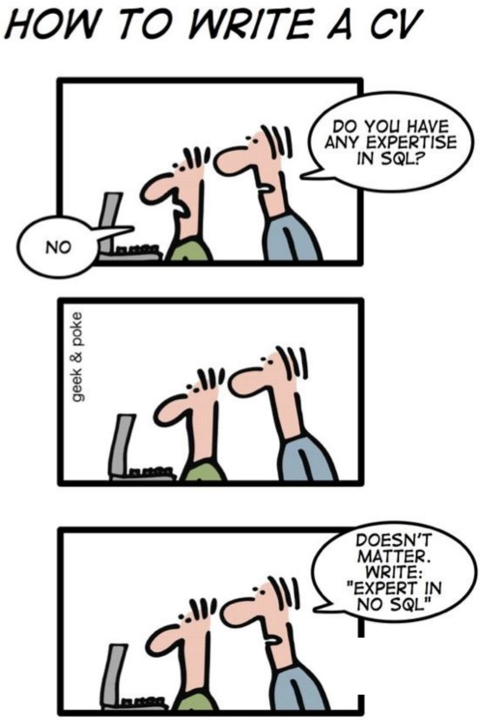
- For this reason internet scale applications moved to distributed file systems
  - Google was the first
- This allowed the data to be partitioned across nodes more efficiently
  - We’ll talk about distributed databases in more details during next lecture

NoSQL vs RDBMS

- Explicit vs Implicit Schema
  - NoSQL databases do have an implicit schema – at least in most cases
- **Relational databases are faster than their NoSQL counterparts for joins, queries, updates, etc**. On the other hand, NoSQL databases are specifically designed for unstructured data (e.g., document-oriented, column-oriented, graph-based, etc).
- **NoSQL databases have flexible data models, scale horizontally, have incredibly fast queries, and are easy for developers to work with**. NoSQL databases typically have very flexible schemas.

What is NOSQL?

from "Geek and Poke"

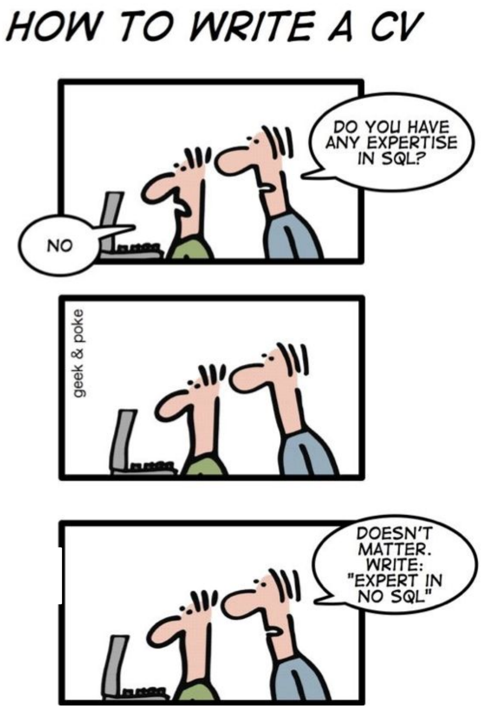


Leverage the NoSQL boom

What is NOSQL?

A NoSQL database provides a mechanism for storage and retrieval of data that use looser consistency models than traditional relational databases in order to achieve horizontal scaling and higher availability.

- NoSWL - "Not only SQL" to emphasize that NoSQL systems do allow SQL-like query language to be used.



Early Applications

- In the early days of Google they made some critical decisions
- Their major architectural drivers were <u>scalability</u> and <u>availability</u>
- They recognized that at some point they would need to scale out
- As they began to scale out they would eventually experience failures
  - If you have enough machines you will regularly experience failures – even with highly reliable machines
- Commodity Hardware:
  - They decided to use cheap machines that were not reliable
  - They had to design for failure regardless: it was advantageous to use many cheap machines

General Approach

- Again, remember that <u>scalability, latency, and availability</u> were important
- It was recognized that <u>hops across nodes were expensive</u>
  - Think joins with a sharded database
- Thus there was an attempt to create a model that stored related data together
- They optimized on the expected usage patterns
  - Fast reads
  - Append operations
  - Modifying existing data is difficult

Main features of NOSQL

1. **Scalability is crucial!**
   - ➢ **Horizontal scalability:** the system is expanded by adding more nodes for data storage and processing as the volume of data grows.
2. **Availability, Replication and Eventual Consistency:** Many applications that use NOSQL systems require continuous system availability.
3. **Replication models:**
   - ➢ **Master-slave replication:** one copy is the master copy; all write operations must be applied to the master copy and then propagated to the slave copies, usually using eventual consistency
   - ➢ **Master-master replication:** reads and writes at any of the replicas but may not guarantee that reads at nodes that store different copies see the same values.

Main features of NOSQL (2)

4. Sharding of Files: horizontal partitioning of file records. Combination of sharding the file records and replicating the shards works in tandem to improve load balancing and data availability.

5. High-Performance Data Access:

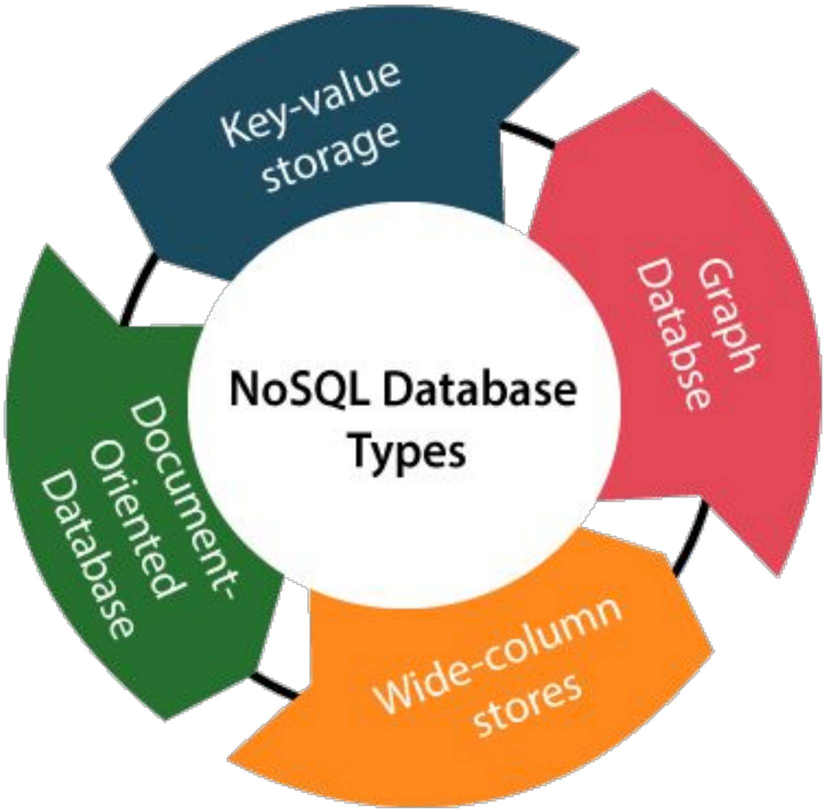
- Hashing**:** hash function h(K) is applied to the key K, and the location of the object with key K is determined by the value of h(K).
- Range partitioning on object keys: the location is determined via a range of key values; for example, location i would hold the objects whose key values K are in the range Ki_min ≤ K ≤ Ki_max.

6. Not Requiring a Schema: allowing semi-structured, self-describing data

7. Less Powerful Query Languages: NOSQL systems typically provide a set of functions and operations as a programming API (e.g., CRUD Operations.)

8. Versioning: Some NOSQL systems provide storage of multiple versions of the data items, with the timestamps of when the data version was created.

NoSQL Databases



Key Value Structures

- Basically you have a key that maps to some “value”
- This value is just a blob
    - <u>The database doesn’t care about the content or structure of this value</u>
- The operations are quite simple e.g.
    - Read (get the value given a key)
    - Insert (inserts a key/value pair)
    - Remove (removes the value associated with a given key)
- Used when you have simple applications that need to store simple objects temporarily.

Document Centric Databases

- Stores a “document”

ID : 123
Customer : 8790
Line Items : [{product id: 2, quantity: 2}
{product id: 34, quantity 1}]
…

Document Centric Databases

- No schema
- You can query the data store
  - Can return all or part of the document
  - Typically query the store by using the id (or key)
- Discussing concurrency only makes sense at the level of a single document

MongoDB Data Model

-Data are organized in **collections.** A collection stores a set of **documents**.

-Collection like table and document like record

  -but: each document can have a different set of attributes even in the same collection

  -Semi-structured schema!

-Only requirement: every document should have an **"_id"** field

MongoDB Example

```
{       "_id”:ObjectId("4efa8d2b7d284dad101e4bc9"),
       "Last Name": ” Cousteau",
       "First Name": ” Jacques-Yves",
       "Date of Birth": ”06-1-1910" },

  {     "_id": ObjectId("4efa8d2b7d284dad101e4bc7"),
      "Last Name": "PELLERIN",
      "First Name": "Franck",
      "Date of Birth": "09-19-1983",
      "Address": "1 chemin des Loges",
      "City": "VERSAILLES" }
```

MongoDB Key Features

- **JSON-style documents**
  - actually uses BSON (JSON's binary format)
- **replication for high availability**
- **auto-sharding for scalability**
- **document-based queries**
- **can create an index on any attribute**
  - for faster reads

MongoDB Terminology

relational term <== >MongoDB equivalent

----

database <== > database

table <== > collection

row <== > document

attributes <== > fields (field-name:value pairs)

primary key <== > the _id field, which is the key associated with the document

MongoDB _id FIeld

Every MongoDB document must have an _id field.

- its value must be unique within the collection
- acts as the primary key of the collection
- it is the key in the key/value pair

- If you create a document without an _id field:
  - MongoDB adds the field for you
  - assigns it a unique BSON ObjectID
  - example from the MongoDB shell:

```
> db.test.save({ rating: "PG-13" })
> db.test.find() { "_id" :ObjectId("528bf38ce6d3df97b49a0569"), "rating" : "PG-13"
}
```

- Note: quoting field names is optional (see rating above)

Data Modeling in MongoDB

Need to determine how to map

entities and relationships => collections of documents

- Could in theory give each type of entity:
  - its own (flexibly formatted) type of document
  - those documents would be stored in the same collection
- However, it can make sense to group different types of entities together.
  - create an aggregate containing data that tends to be accessed together

(a) project document with an array of embedded workers:

```
{
    _id:            "P1",
    Pname:          "ProductX",
    Plocation:      "Bellaire",
    Workers: [
                { Ename: "John Smith",
                  Hours: 32.5
                },
                { Ename: "Joyce English",
                  Hours: 20.0
                }
             ]
);
```

Example of simple documents in MongoDB.

(a) Denormalized document design with embedded subdocuments.

(a) project document with an array of embedded workers:

```
{
    _id:            "P1",
    Pname:          "ProductX",
    Plocation:      "Bellaire",
    Workers: [
                { Ename: "John Smith",
                  Hours: 32.5
                },
                { Ename: "Joyce English",
                  Hours: 20.0
                }
             ]
);
```

Example of simple documents in MongoDB.
(a) Denormalized document design with embedded subdocuments.

(b) project document with an embedded array of worker ids:

```
{
    _id:          "P1",
    Pname:        "ProductX",
    Plocation:    "Bellaire",
    WorkerIds:    [ "W1", "W2" ]
}
    { _id:        "W1",
    Ename:        "John Smith",
    Hours:        32.5
}
    { _id:        "W2",
    Ename:        "Joyce English",
    Hours:        20.0
}
```

Example of simple documents in MongoDB.
(a) Embedded array of document references.

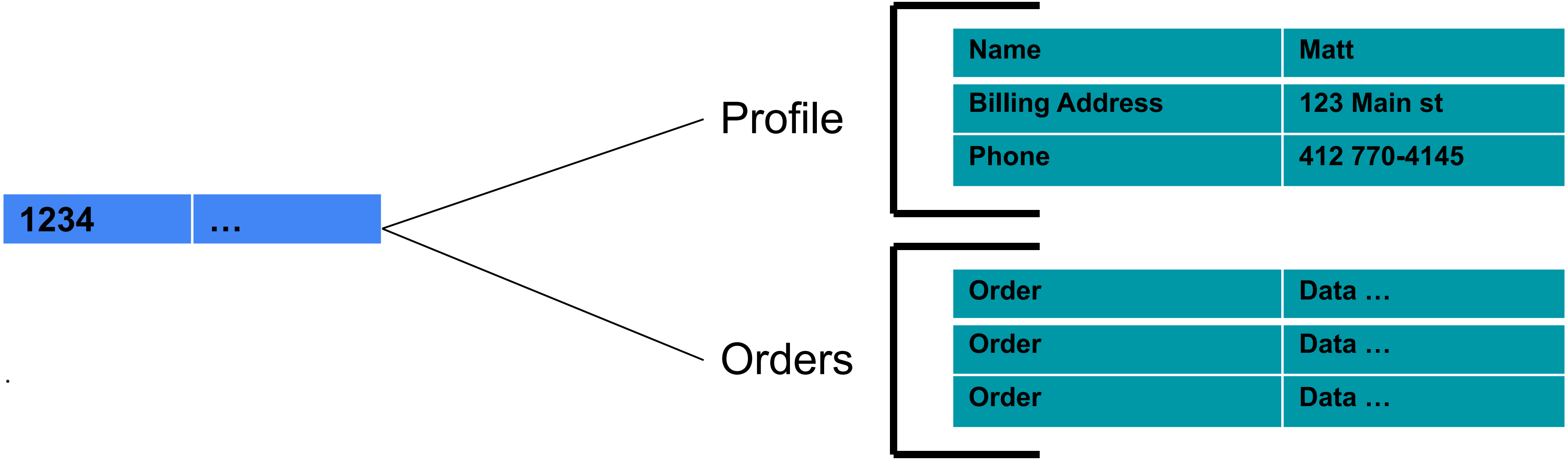
(c) normalized project and worker documents (not a fully normalized design for M:N relationships):

```
{
    _id:            "P1",
    Pname:          "ProductX",
    Plocation:      "Bellaire"
}
{   _id:            "W1",
    Ename:          "John Smith",
    ProjectId:      "P1",
    Hours:          32.5
}
```

Example of simple documents in MongoDB.
(a) Normalized documents.

Column Databases

- Row key maps to “column families”
  - similar to the data represented in relational databases.



.

- **It is used for a large dataset that can be distributed across multiple database nodes, especially when the columns are not always the same for every row.**

Examples

Key Value

- DynamoDB
- Azure Table
- Redis
- Riak

Document Centric

- MongoDB
- CouchDB
- RavenDB

Column

- Hbase
- Cassandra
- Hypertable
- SimpleDB

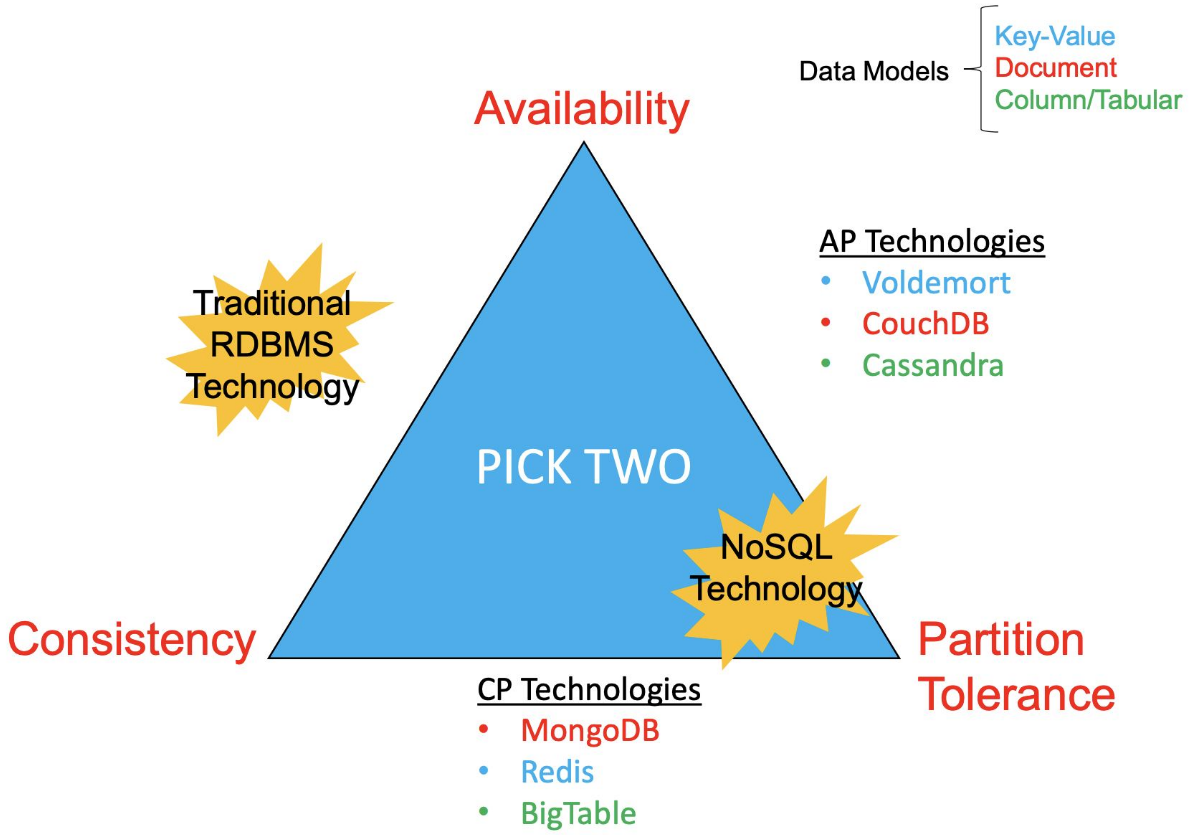
ACID Properties

- **Atomicity**: A transaction is an atomic unit of processing; it is either performed in its entirety or not performed at all.
- **Consistency preservation**: 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; this property, when enforced strictly, solves the temporary update problem and makes cascading rollbacks of transactions unnecessary.
- **Durability or permanency**: Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failure.

Consistency - CAP Theorem

- When data becomes distributed you need to worry about a network partition
  - Essentially this means that instances of your data store can’t communicate
- When this happens you need to choose between availability or consistency

Consistency - CAP Theorem



Reading Material

- Fundamentals of Database Systems. Ramez Elmasri and Shamkant B. Navathe. Pearson. **Chapter 21, Chapter 16 and Chapter 24.**


Q & A