



**RV College of
Engineering®**

Go, change the world

Unit 5 (Recovery Concepts & NoSql)

Original Content:

Ramez Elmasri and Shamkant B. Navathe

Dr. G. Shobha

Professor, CSE Department

RV College of Engineering, Bengaluru - 59

PRESENTATION CONTENTS

- Recovery algorithms
- Recovery concepts
 - Write-ahead logging
 - In-place versus shadow updates
 - Rollback
 - Deferred update
 - Immediate update
- Certain recovery techniques best used with specific concurrency control methods

Recovery Concepts

- Recovery process restores database to most recent consistent state before time of failure
- Information kept in system log

Typical recovery strategies

- Restore backed-up copy of database

Best in cases of extensive damage

- Identify any changes that may cause inconsistency

Best in cases of noncatastrophic failure

Some operations may require redo

Recovery Concepts (cont'd.)

- Deferred update techniques
 - Do not physically update the database until after transaction commits
 - Undo is not needed; redo may be needed
- Immediate update techniques
 - Database may be updated by some operations of a transaction before it reaches commit point
 - Operations also recorded in log
 - Recovery still possible

Recovery Concepts (cont'd.)

- Undo and redo operations required to be idempotent
 - Executing operations multiple times equivalent to executing just once
 - Entire recovery process should be idempotent
- Caching (buffering) of disk blocks
 - DBMS cache: a collection of in-memory buffers
 - Cache directory keeps track of which database items are in the buffers

Recovery Concepts (cont'd.)

- Cache buffers replaced (flushed) to make space for new items
- Dirty bit associated with each buffer in the cache
 - Indicates whether the buffer has been modified
- Contents written back to disk before flush if dirty bit equals one
- Pin-unpin bit
 - Page is pinned if it cannot be written back to disk yet

Recovery Concepts (cont'd.)

- Main strategies
 - In-place updating
 - Writes the buffer to the same original disk location
 - Overwrites old values of any changed data items
 - Shadowing
 - Writes an updated buffer at a different disk location, to maintain multiple versions of data items
 - Not typically used in practice
- Before-image: old value of data item
- After-image: new value of data item

Recovery Concepts (cont'd.)

- Write-ahead logging
 - Ensure the before-image (BFIM) is recorded
 - Appropriate log entry flushed to disk
 - Necessary for UNDO operation if needed
- UNDO-type log entries
- REDO-type log entries

Recovery Concepts (cont'd.)

- Steal/no-steal and force/no-force
 - Specify rules that govern when a page from the database cache can be written to disk
- No-steal approach
 - Cache buffer page updated by a transaction cannot be written to disk before the transaction commits
- Steal approach
 - Recovery protocol allows writing an updated buffer before the transaction commits

Recovery Concepts (cont'd.)

- Force approach
 - All pages updated by a transaction are immediately written to disk before the transaction commits
 - Otherwise, no-force
- Typical database systems employ a steal/no-force strategy
 - Avoids need for very large buffer space
 - Reduces disk I/O operations for heavily updated pages

Recovery Concepts (cont'd.)

- Write-ahead logging protocol for recovery algorithm requiring both UNDO and REDO
 - BFIM of an item cannot be overwritten by its after image until all UNDO-type log entries have been force-written to disk
 - Commit operation of a transaction cannot be completed until all REDO-type and UNDO-type log records for that transaction have been force-written to disk

Checkpoints in the System Log and Fuzzy Checkpointing

- Taking a checkpoint
 - Suspend execution of all transactions temporarily
 - Force-write all main memory buffers that have been modified to disk
 - Write a checkpoint record to the log, and force-write the log to the disk
 - Resume executing transactions
- DBMS recovery manager decides on checkpoint interval

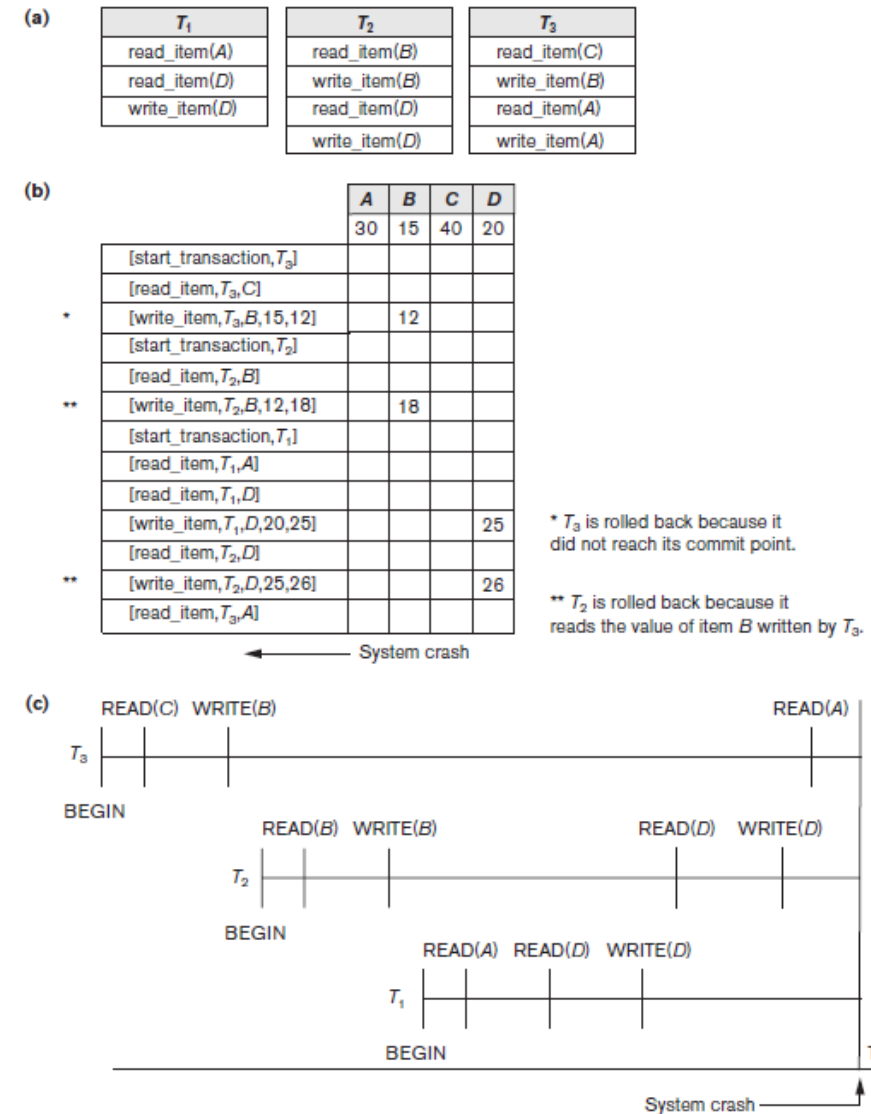
Checkpoints in the System Log and Fuzzy Checkpointing (cont'd.)

- Fuzzy checkpointing
 - System can resume transaction processing after a `begin_checkpoint` record is written to the log
 - Previous checkpoint record maintained until `end_checkpoint` record is written

Transaction Rollback

- Transaction failure after update but before commit
 - Necessary to roll back the transaction
 - Old data values restored using undo-type log entries
- Cascading rollback
 - If transaction T is rolled back, any transaction S that has read value of item written by T must also be rolled back
 - Almost all recovery mechanisms designed to avoid this

Figure 22.1 Illustrating cascading rollback (a process that never occurs in strict or cascadeless schedules) (a) The read and write operations of three transactions (b) System log at point of crash (c) Operations before the crash



Transactions that Do Not Affect the Database

- Example actions: generating and printing messages and reports
- If transaction fails before completion, may not want user to get these reports
 - Reports should be generated only after transaction reaches commit point
- Commands that generate reports issued as batch jobs executed only after transaction reaches commit point
 - Batch jobs canceled if transaction fails

22.2 NO-UNDO/REDO Recovery Based on Deferred Update

- Deferred update concept
 - Postpone updates to the database on disk until the transaction completes successfully and reaches its commit point
 - Redo-type log entries are needed
 - Undo-type log entries not necessary
 - Can only be used for short transactions and transactions that change few items
 - Buffer space an issue with longer transactions

NO-UNDO/REDO Recovery Based on Deferred Update (cont'd.)

- Deferred update protocol
 - Transaction cannot change the database on disk until it reaches its commit point
 - All buffers changed by the transaction must be pinned until the transaction commits (no-steal policy)
 - Transaction does not reach its commit point until all its REDO-type log entries are recorded in log and log buffer is force-written to disk

NO-UNDO/REDO Recovery Based on Deferred Update (cont'd.)

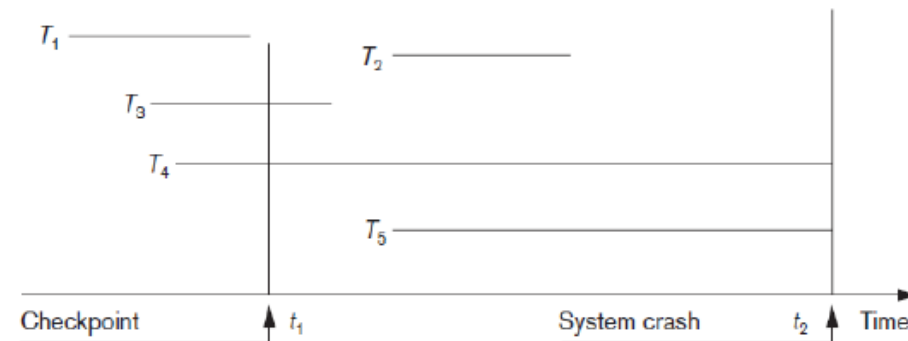


Figure 22.2 An example of a recovery timeline to illustrate the effect of checkpointing

22.3 Recovery Techniques Based on Immediate Update

- Database can be updated immediately
 - No need to wait for transaction to reach commit point
 - Not a requirement that every update be immediate
- UNDO-type log entries must be stored
- Recovery algorithms
 - UNDO/NO-REDO (steal/force strategy)
 - UNDO/REDO (steal/no-force strategy)

T_1	T_2	T_3	T_4
read_item(A)	read_item(B)	read_item(A)	read_item(B)
read_item(D)	write_item(B)	write_item(A)	write_item(B)
write_item(D)	read_item(D)	read_item(C)	read_item(A)
	write_item(D)	write_item(C)	write_item(A)

Figure 22.3 An example of recovery using deferred update with concurrent transactions (a) The READ and WRITE operations of four transactions (b) System log at the point of crash

[start_transaction, T_1]
[write_item, T_1 , D, 20]
[commit, T_1]
[checkpoint]
[start_transaction, T_4]
[write_item, T_4 , B, 15]
[write_item, T_4 , A, 20]
[commit, T_4]
[start_transaction, T_2]
[write_item, T_2 , B, 12]
[start_transaction, T_3]
[write_item, T_3 , A, 30]
[write_item, T_2 , D, 25]

← System crash

T_2 and T_3 are ignored because they did not reach their commit points.

T_4 is redone because its commit point is after the last system checkpoint.

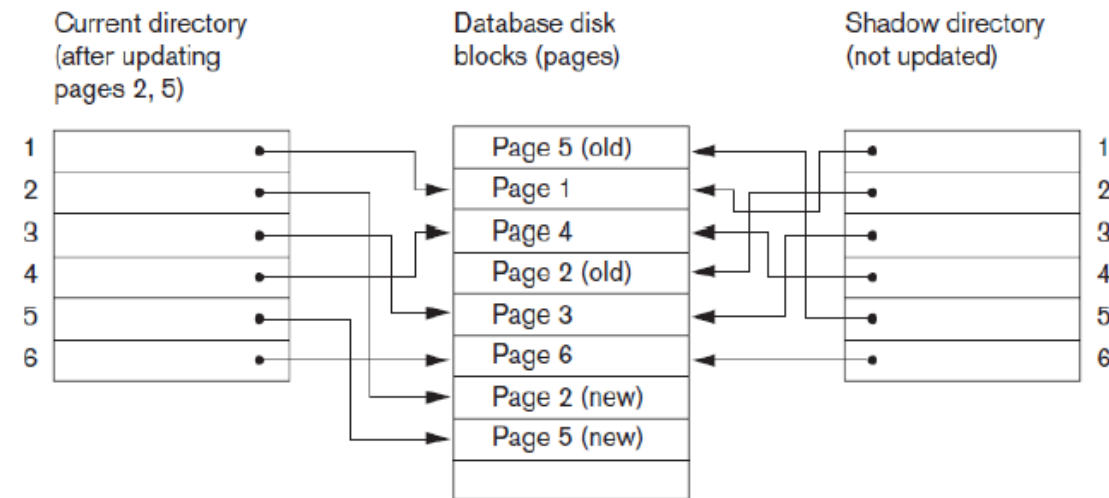
22.4 Shadow Paging

- No log required in a single-user environment
 - Log may be needed in a multiuser environment for the concurrency control method
- Shadow paging considers disk to be made of n fixed-size disk pages
 - Directory with n entries is constructed
 - When transaction begins executing, directory copied into shadow directory to save while current directory is being used
 - Shadow directory is never modified

Shadow Paging (cont'd.)

- New copy of the modified page created and stored elsewhere
 - Current directory modified to point to new disk block
 - Shadow directory still points to old disk block
- Failure recovery
 - Discard current directory
 - Free modified database pages
 - NO-UNDO/NO-REDO technique

Shadow Paging (cont'd.)



⁵The directory is similar to the page table maintained by the operating system for each process.

Figure 22.4 An example of shadow paging

22.5 The ARIES Recovery Algorithm

- Used in many IBM relational database products
- Uses a steal/no-force approach for writing
- Concepts
 - Write-ahead logging
 - Repeating history during redo
 - Retrace all database system actions prior to crash to reconstruct database state when crash occurred
 - Logging changes during undo
 - Prevents ARIES from repeating completed undo operations if failure occurs during recovery

The ARIES Recovery Algorithm (cont'd.)

- Analysis step
 - Identifies dirty (updated) pages in the buffer and set of transactions active at the time of crash
 - Determines appropriate start point in the log for the REDO operation
- REDO
 - Reapplies updates from the log to the database
 - Only necessary REDO operations are applied

The ARIES Recovery Algorithm (cont'd.)

- UNDO
 - Log is scanned backward
 - Operations of transactions that were active at the time of the crash are undone in reverse order
- Every log record has associated log sequence number (LSN)
 - Indicates address of log record on disk
 - Corresponds to a specific change of some transaction

ARIES Recovery Example

Figure 22.5 An example of recovery in ARIES (a) The log at point of crash (b) The Transaction and Dirty Page Tables at time of checkpoint (c) The Transaction and Dirty Page Tables after the analysis phase

(a)	Lsn	Last_lsn	Tran_id	Type	Page_id	Other_information
	1	0	T_1	update	C	...
	2	0	T_2	update	B	...
	3	1	T_1	commit		...
	4	begin checkpoint				
	5	end checkpoint				
	6	0	T_3	update	A	...
	7	2	T_2	update	C	...
	8	7	T_2	commit		...

(b)	TRANSACTION TABLE			DIRTY PAGE TABLE	
	Transaction_id	Last_lsn	Status	Page_id	Lsn
	T_1	3	commit	C	1
	T_2	2	in progress	B	2

(c)	TRANSACTION TABLE			DIRTY PAGE TABLE	
	Transaction_id	Last_lsn	Status	Page_id	Lsn
	T_1	3	commit	C	7
	T_2	8	commit	B	2
	T_3	6	in progress	A	6



**RV College of
Engineering®**

Go, change the world

UNIT 5

NoSql

*Dr. G. Shobha
Professor, CSE Department
RV College of Engineering, Bengaluru - 59*

NoSQL



- Term appeared in the late 90s
 - open-source relational database [Strozzi NoSQL]
 - tables as ASCII files, without SQL
- Current interpretation
 - June 11, 2009: meetup in San Francisco
 - Open-source, distributed, non-relational databases
 - Hashtag chosen: #NoSQL
 - Main features:
 - Not using SQL and the relational model
 - Open-source projects (mostly)
 - Running on clusters
 - Schemaless
 - However, no accepted precise definitions
- Most people say that NoSQL means "Not Only SQL"



Key Points

- Relational databases have been a successful technology for twenty years, providing persistence, concurrency control, and an integration mechanism
- Application developers have been frustrated with the impedance mismatch between the relational model and the in-memory data structures
- There is a movement away from using databases as integration points towards encapsulating databases within applications and integrating through services
- The vital factor for a change in data storage was the need to support large volumes of data by running on clusters. Relational databases are not designed to run efficiently on clusters.
- NoSQL is an accidental neologism. There is no prescriptive definition—all you can make is an observation of common characteristics.
- The common characteristics of NoSQL databases are:
 - Not using the relational model
 - Running well on clusters
 - Open-source
 - Schemaless

Popularity

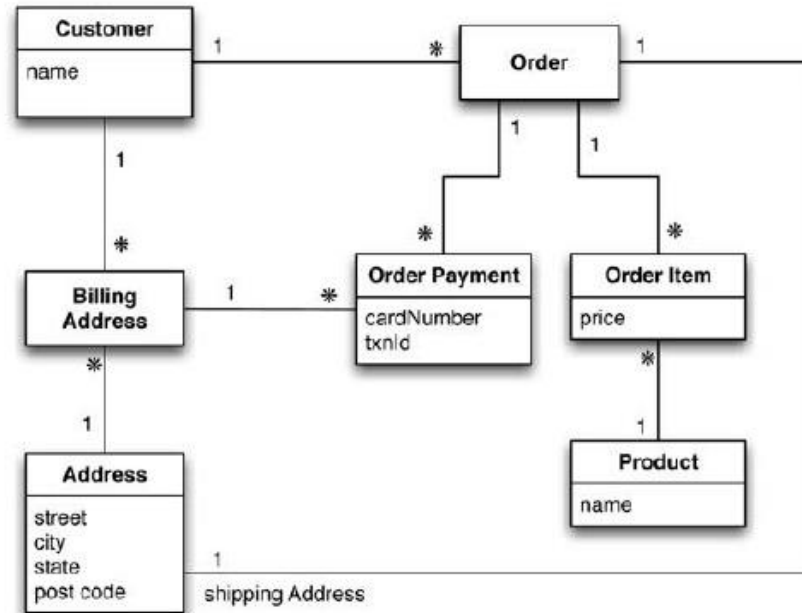
NoSQL Data Models

- A data model is a set of constructs for representing the information
 - Relational model: tables, columns and rows
- Storage model: how the DBMS stores and manipulates the data internally
- A data model is usually independent of the storage model
- Data models for NoSQL systems:
 - aggregate models
 - key-value,
 - document,
 - column-family
 - graph-based models

Aggregates

- Data as atomic units that have a complex structure
 - more structure than just a set of tuples
 - example:
 - complex record with: simple fields, arrays, records nested inside
- Aggregate in Domain-Driven Design
 - a collection of related objects that we treat as a unit
 - a unit for data manipulation and management of consistency
- Advantages of aggregates:
 - easier for application programmers to work with
 - easier for database systems to handle operating on a cluster

Example



Relational implementation

Customer	
Id	Name
1	Martin

Orders		
Id	CustomerId	ShippingAddressId
99	1	77

Product	
Id	Name
27	NoSQL Distilled

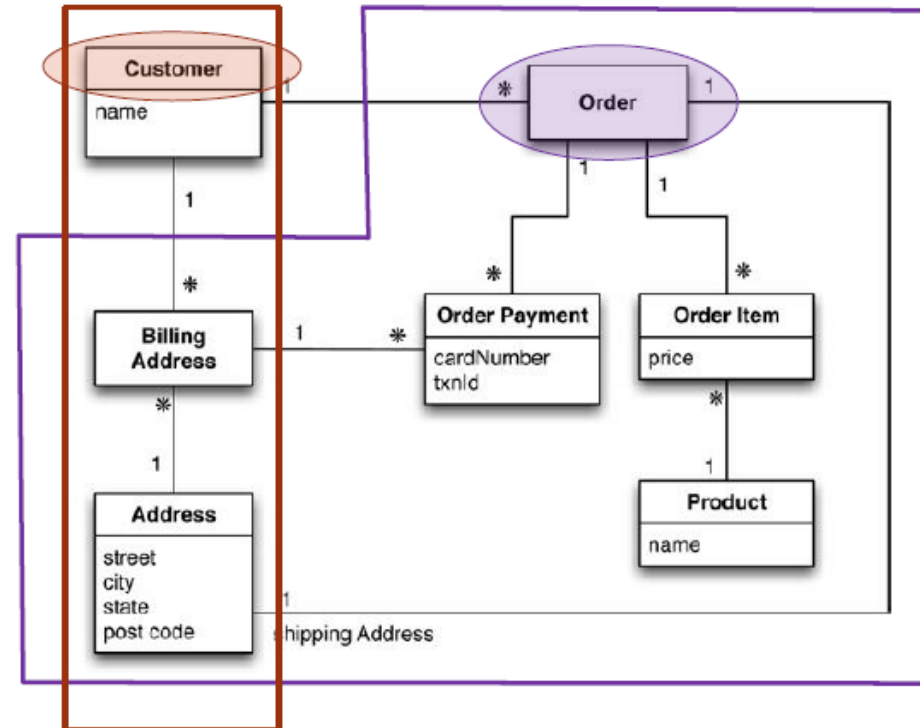
BillingAddress		
Id	CustomerId	AddressId
55	1	77

OrderItem			
Id	OrderId	ProductId	Price
100	99	27	32.45

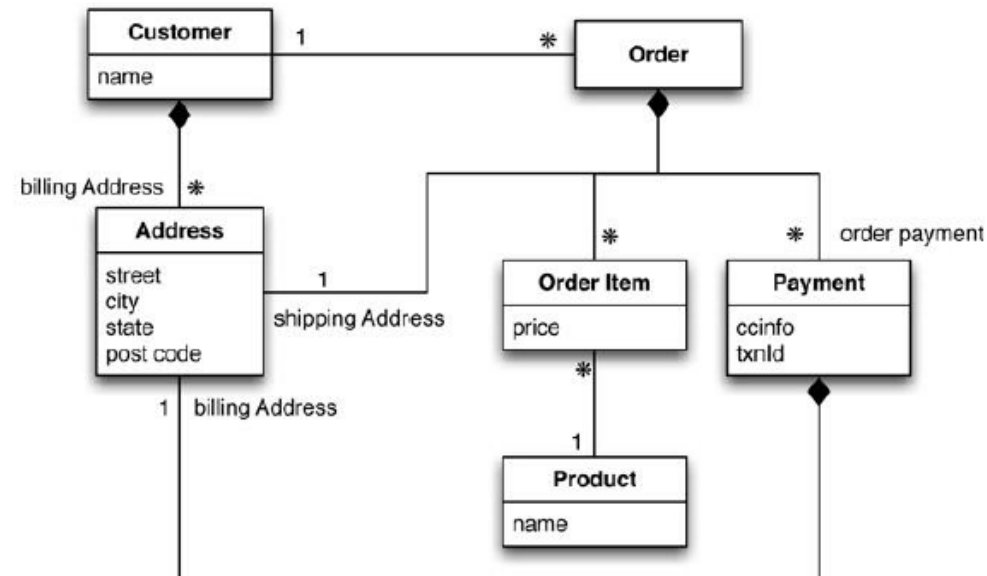
Address	
Id	City
77	Chicago

OrderPayment				
Id	OrderId	CardNumber	BillingAddressId	txnId
33	99	1000-1000	55	abeli*879rft

A possible aggregation



Aggregate representation

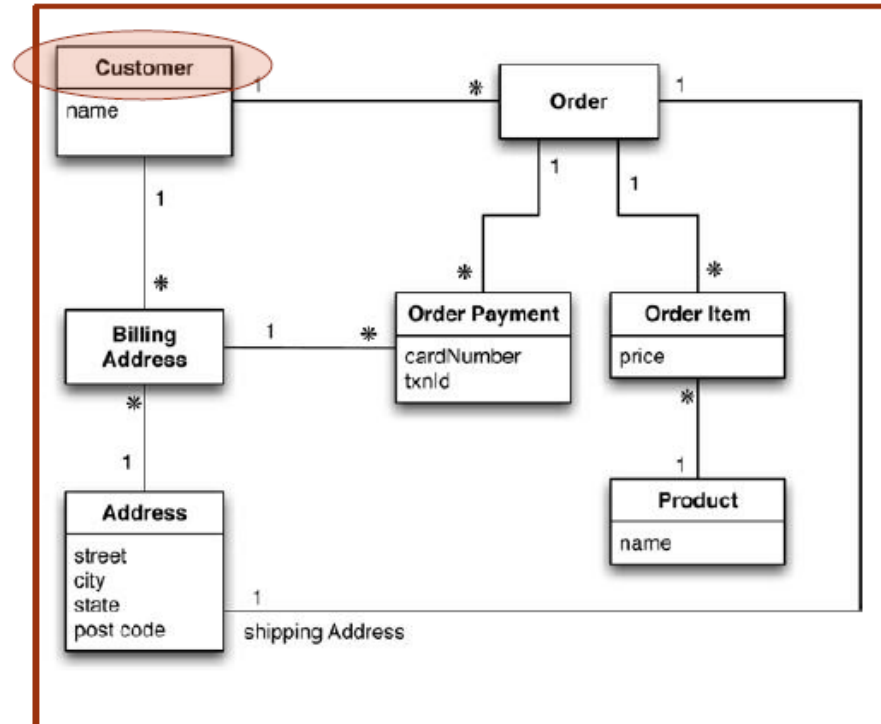


Aggregate implementation

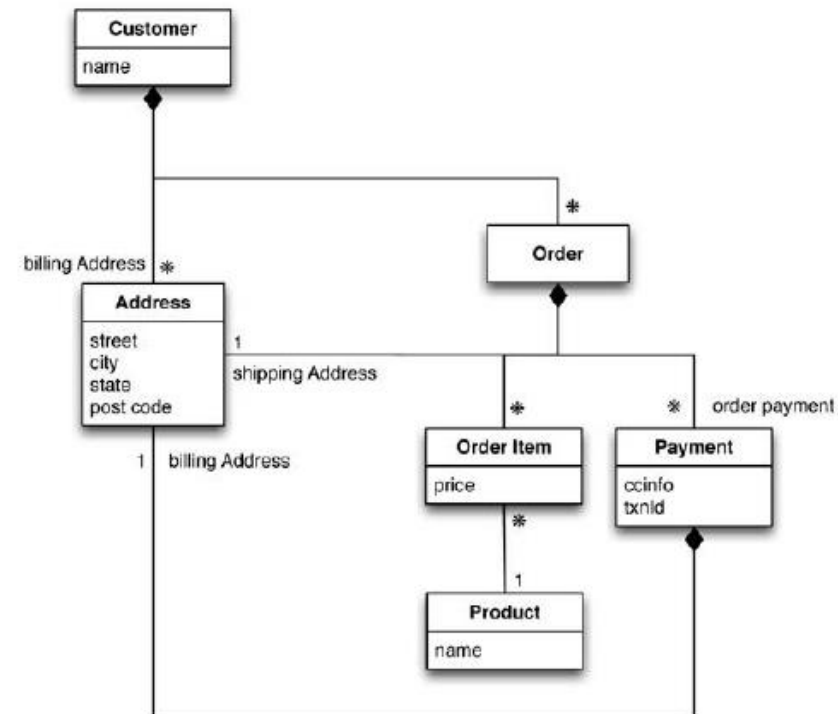
```
// in customers
{
  "id":1,
  "name":"Martin",
  "billingAddress":[{"city":"Chicago"}]
}

// in orders
{
  "id":99,
  "customerId":1,
  "orderItems":[
    {
      "productId":27,
      "price": 32.45,
      "productName": "NoSQL Distilled"
    }
  ],
  "shippingAddress":[{"city":"Chicago"}]
  "orderPayment":[
    {
      "ccinfo":"1000-1000-1000-1000",
      "txnId":"abelif879rft",
      "billingAddress": {"city": "Chicago"}
    }
  ],
}
```

Another possible aggregation



Aggregate representation (2)



Aggregate implementation (2)

```
// in customers
{
  "customer": {
    "id": 1,
    "name": "Martin",
    "billingAddress": [{"city": "Chicago"}],
    "orders": [
      {
        "id": 99,
        "customerId": 1,
        "orderItems": [
          {
            "productId": 27,
            "price": 32.45,
            "productName": "NoSQL Distilled"
          }
        ]
      },
      {
        "id": 100,
        "customerId": 1,
        "orderItems": [
          {
            "productId": 28,
            "price": 29.99,
            "productName": "NoSQL Distilled"
          }
        ]
      }
    ],
    "shippingAddress": [{"city": "Chicago"}],
    "orderPayment": [
      {
        "ccinfo": "1000-1000-1000-1000",
        "txnId": "abelif879rft",
        "billingAddress": {"city": "Chicago"}
      }
    ]
  }
}
```

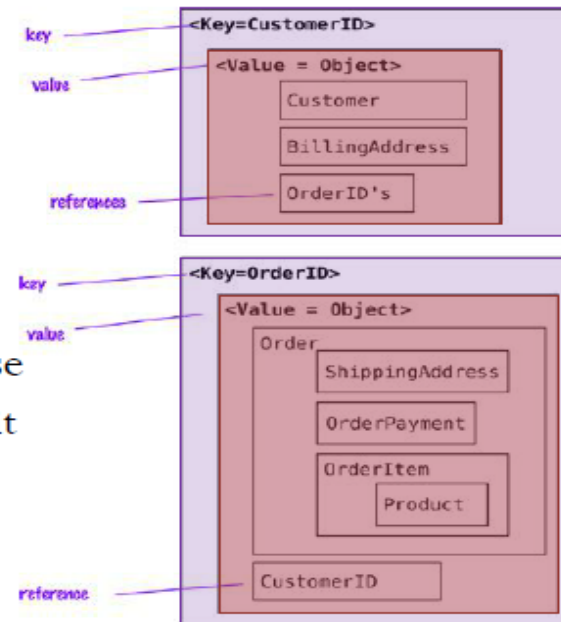
Design strategy

- No universal answer for how to draw aggregate boundaries
- It depends entirely on how you tend to manipulate data!
 - Accesses on a single order at a time: first solution
 - Accesses on customers with all orders: second solution
- Context-specific
 - some applications will prefer one or the other
 - even within a single system
- Focus on the unit of interaction with the data storage
- Pros:
 - it helps greatly with running on a cluster: data will be manipulated together, and thus should live on the same node!
- Cons:
 - an aggregate structure may help with some data interactions but be an obstacle for others



Key-Value Databases

- Strongly aggregate-oriented
 - Lots of aggregates
 - Each aggregate has a key
- Data model:
 - A set of $\langle \text{key}, \text{value} \rangle$ pairs
 - Value: an aggregate instance
- The aggregate is **opaque** to the database
 - just a big blob of mostly meaningless bit
- Access to an aggregate:
 - lookup based on its key



Popular key-value databases



Azure Cosmos DB



Document databases

- Strongly aggregate-oriented
 - Lots of aggregates
 - Each aggregate has a key
- Data model:
 - A set of <key,document> pairs
 - Document: an aggregate instance
- Structure of the aggregate **visible**
 - limits on what we can place in it
- Access to an aggregate:
 - queries based on the fields in the aggregate

```
# Customer object
{
  "customerId": 1,
  "name": "Martin",
  "billingAddress": [{"city": "Chicago"}],
  "payment": [
    {"type": "debit",
     "ccinfo": "1000-1000-1000-1000"}
  ]
}
```

```
# Order object
{
  "orderId": 99,
  "customerId": 1,
  "orderDate": "Nov-20-2011",
  "orderItems": [{"productId": 27, "price": 32.45}],
  "orderPayment": [{"ccinfo": "1000-1000-1000-1000",
    "txnId": "abelif879rft"}],
  "shippingAddress": {"city": "Chicago"}
}
```

Popular document databases



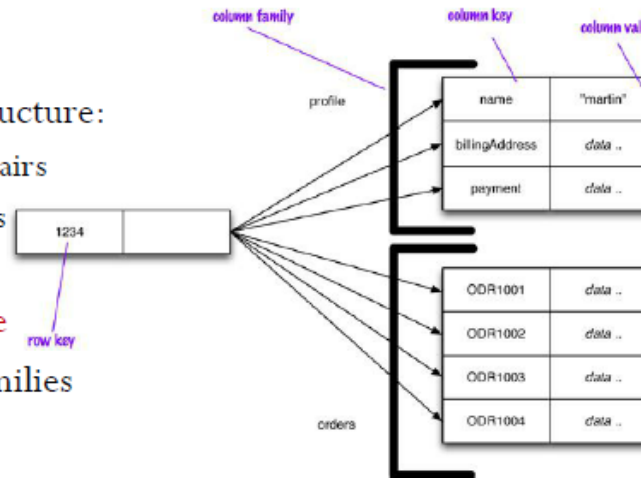
Azure Cosmos DB

Key-Value vs Document stores

- Key-value database
 - A key plus a big blob of mostly meaningless bits
 - We can store whatever we like in the aggregate
 - We can only access an aggregate by lookup based on its key
- Document database
 - A key plus a structured aggregate
 - More flexibility in access
 - we can submit queries to the database based on the fields in the aggregate
 - we can retrieve part of the aggregate rather than the whole thing
 - Indexes based on the contents of the aggregate

Column(-Family) Stores

- Strongly aggregate-oriented
 - Lots of aggregates
 - Each aggregate has a key
- Data model: a two-level map structure:
 - A set of <row-key, aggregate> pairs
 - Each aggregate is a group of pairs <column-key, value>
- Structure of the aggregate **visible**
- Columns can be organized in families
 - Data usually accessed together
- Access to an aggregate:
 - accessing the row as a whole
 - picking out a particular column



Properties of Column Stores

- Operations also allow picking out a particular column
 - `get('1234', 'name')`
- Each column:
 - has to be part of a single column family
 - acts as unit for access
- You can add any column to any row, and rows can have very different columns
- You can model a list of items by making each item a separate column.
- Two ways to look at data:
 - Row-oriented
 - Each row is an aggregate
 - Column families represent useful chunks of data within that aggregate.
 - Column-oriented:
 - Each column family defines a record type
 - Row as the join of records in all column families

Cassandra



- Skinny row
 - few columns
 - same columns used by many different rows
 - each row is a record and each column is a field
- Wide row
 - many columns (perhaps thousands)
 - rows having very different columns
 - models a list, with each column being one element in that list
- A column store can contain both field-like columns and list-like columns

Popular column stores



Key Points

- An aggregate is a collection of data that we interact with as a unit.
- Aggregates form the boundaries for ACID operations with the database
- Key-value, document, and column-family databases can all be seen as forms of aggregate-oriented database
- Aggregates make it easier for the database to manage data storage over clusters
- Aggregate-oriented databases work best when most data interaction is done with the same aggregate
- Aggregate-ignorant databases are better when interactions use data organized in many different formations

Relationships

- Relationship between different aggregates:
 - Put the ID of one aggregate within the data of the other
 - Join: write a program that uses the ID to link data
 - The database is ignorant of the relationship in the data

```
// in customers
{
  "id":1,
  "name":"Martin",
  "billingAddress":[{"city":"Chicago"}]
}
```

```
// in orders
{
  "id":99,
  "customerId":1,
  "orderItems":[
    {
      "productId":27,
      "price": 32.45,
      "productName": "NoSQL Distilled"
    }
  ],
  "shippingAddress":[{"city":"Chicago"}]
  "orderPayment":[
    {
      "ccinfo":"1000-1000-1000-1000",
      "txnId":"abelif879rft",
      "billingAddress": {"city": "Chicago"}
    }
  ],
}
```

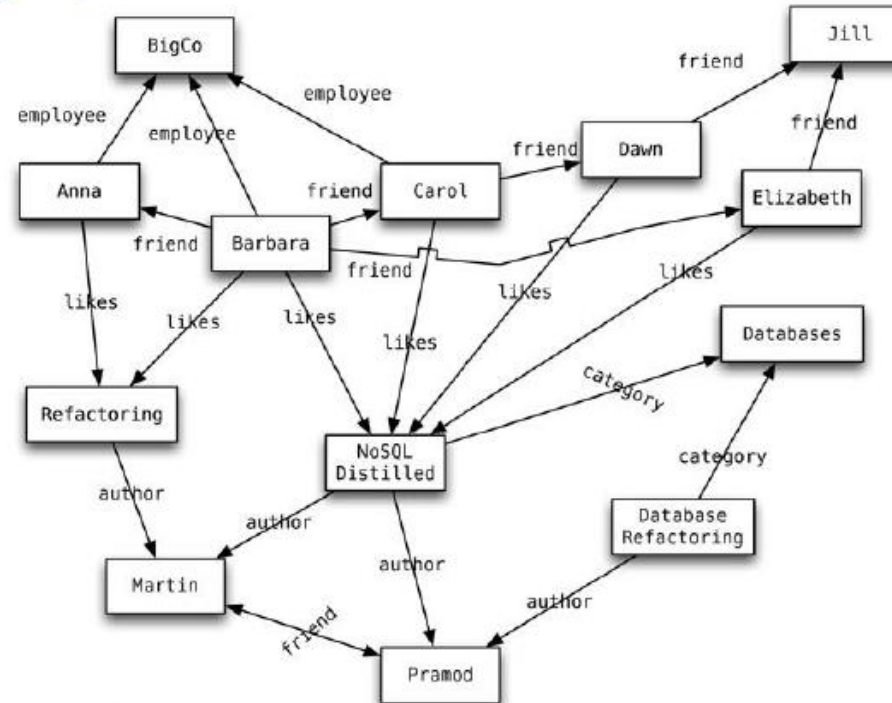
Relationship management

- Many NoSQL databases provide ways to make relationships visible to the database
 - Document stores makes use of indexes
 - Riak (key-value store) allows you to put link information in metadata
- But what about updates?
 - Aggregate-oriented databases treat the aggregate as the unit of data-retrieval.
 - Atomicity is only supported within the contents of a single aggregate.
 - Updates over multiple aggregates at once is a programmer's responsibility!
 - In contrast, relational databases provide ACID guarantees while altering many rows through transactions

Graph Databases

- Graph databases are motivated by a different frustration with relational databases
 - Complex relationships require complex join
- Goal:
 - Capture data consisting of complex relationships
 - Data naturally modelled as graphs
 - Examples: Social networks, Web data, maps, networks.

A graph database



Possible query: “find the authors of books in the Databases category that a friend of mine likes.”

Popular graph databases



Data model of graph databases

- Basic characteristic: nodes connected by edges (also called arcs).
- Beyond this: a lot of variation in data models
 - Neo4J stores Java objects to nodes and edges in a schemaless fashion
 - InfiniteGraph stores Java objects, which are subclasses of built-in types, as nodes and edges.
 - FlockDB is simply nodes and edges with no mechanism for additional attributes
- Queries
 - Navigation through the network of edges
 - You do need a starting place
 - Nodes can be indexed by an attribute such as ID.

Graph vs Relational databases

- Relational databases
 - implement relationships using foreign keys
 - joins require to navigate around and can get quite expensive
- Graph databases
 - make traversal along the relationships very cheap
 - performance is better for highly connected data
 - shift most of the work from query time to insert time
 - good when querying performance is more important than insert speed

Graph vs Aggregate-oriented databases

- Very different data models
- Aggregate-oriented databases
 - distributed across clusters
 - simple query languages
 - no ACID guarantees
- Graph databases
 - more likely to run on a single server
 - graph-based query languages
 - transactions maintain consistency over multiple nodes and edges

Schemaless Databases

- Key-value store allows you to store any data you like under a key
- Document databases make no restrictions on the structure of the documents you store
- Column-family databases allow you to store any data under any column you like
- Graph databases allow you to freely add new edges and freely add properties to nodes and edges as you wish

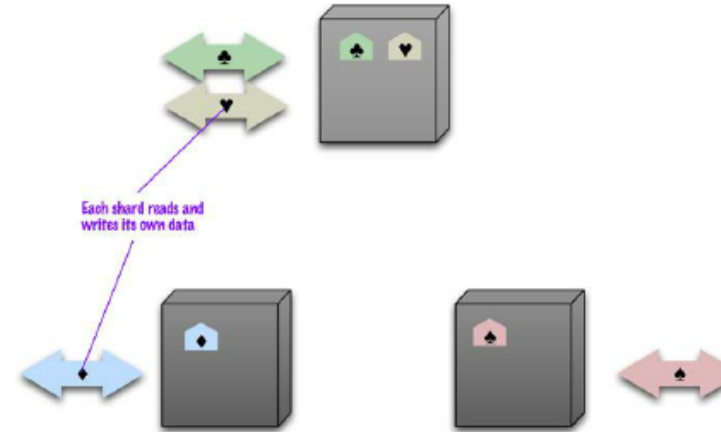
Pros and cons of schemaless data

- Pros:
 - More freedom and flexibility
 - You can easily change your data organization
 - You can deal with non-uniform data
- Cons:
 - A program that accesses data:
 - almost always relies on some form of implicit schema
 - it assumes that certain fields are present
 - The implicit schema is shifted into the application code that accesses data
 - To understand what data is present you have look at the application code
 - The schema cannot be used to:
 - decide how to store and retrieve data efficiently
 - ensure data consistency
 - Problems if multiple applications, developed by different people, access the same database.

Data distribution

- NoSQL systems: data distributed over large clusters
- Aggregate is a natural unit to use for data distribution
- Data distribution models:
 - Single server (is an option for some applications)
 - Multiple servers
- Orthogonal aspects of data distribution:
 - Sharding: different data on different nodes
 - Replication: the same data copied over multiple nodes

Sharding



- Different parts of the data onto different servers
 - Horizontal scalability
 - Ideal case: different users all talking to different server nodes
 - Data accessed together on the same node — aggregate unit!
- Pros: it can improve both reads and writes
- Cons: Clusters use less reliable machines — resilience decreases

Improving performance

Main rules of sharding:

1. Place the data close to where it's accessed
 - Orders for Boston: data in your eastern US data center
 2. Try to keep the load even
 - All nodes should get equal amounts of the load
 3. Put together data that may be read in sequence
 - Same order, same node
- Many NoSQL databases offer **auto-sharding**
 - the database takes on the responsibility of sharding

Replication



It comes in two forms:

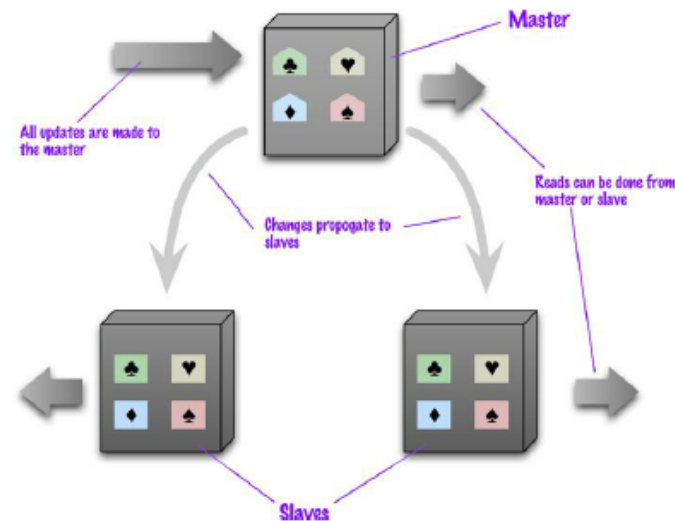


master-slave



peer-to-peer

Master-Slave Replication

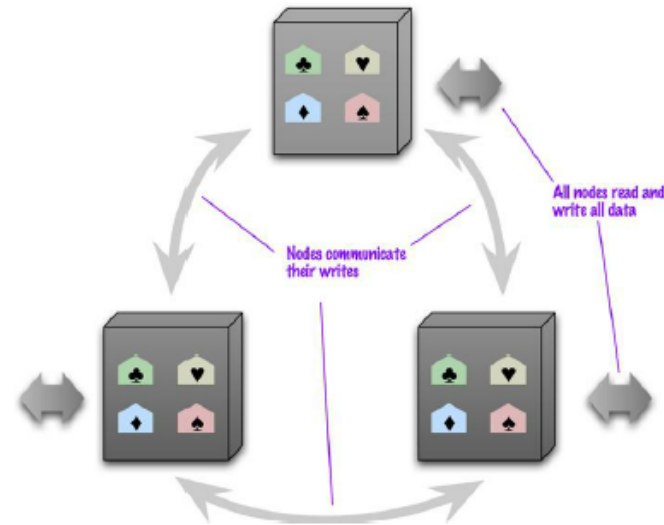


- Master
 - is the authoritative source for the data
 - is responsible for processing any updates to that data
 - can be appointed manually or automatically
- Slaves
 - A replication process synchronizes the slaves with the master
 - After a failure of the master, a slave can be appointed as new master very quickly

Pros and cons of Master-Slave Replication

- Pros
 - More read requests:
 - Add more slave nodes
 - Ensure that all read requests are routed to the slaves
 - Should the master fail, the slaves can still handle read requests
 - Good for datasets with a read-intensive dataset
- Cons
 - The master is a bottleneck
 - Limited by its ability to process updates and to pass those updates on
 - Its failure does eliminate the ability to handle writes until:
 - the master is restored or
 - a new master is appointed
 - Inconsistency due to slow propagation of changes to the slaves
 - Bad for datasets with heavy write traffic

Peer-to-Peer Replication

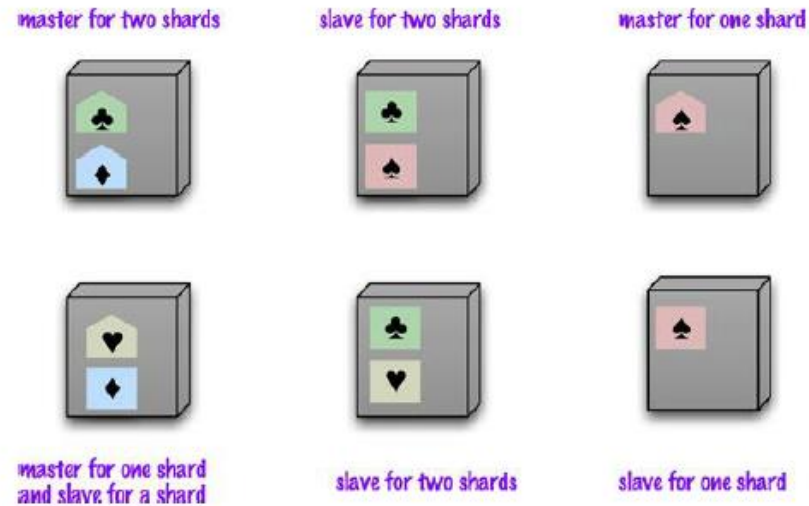


- All the replicas have equal weight, they can all accept writes
- The loss of any of them doesn't prevent access to the data store.

Pros and cons of peer-to-peer replication

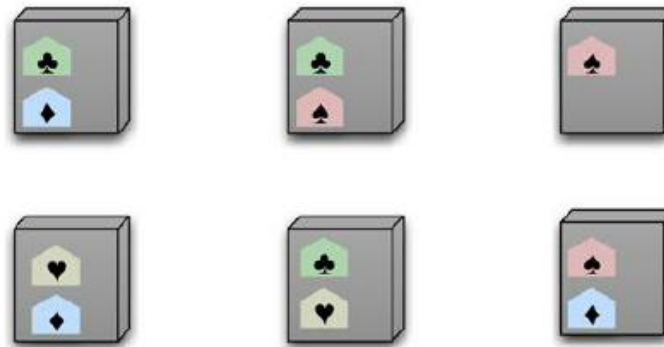
- Pros:
 - you can ride over node failures without losing access to data
 - you can easily add nodes to improve your performance
- Cons:
 - Inconsistency!
 - Slow propagation of changes to copies on different nodes
 - Inconsistencies on read lead to problems but are relatively transient
 - Two people can update different copies of the same record stored on different nodes at the same time - a **write-write conflict**.
 - Inconsistent writes are forever.

Sharding and Replication on MS



- We have multiple masters, but each data only has a single master.
- Two schemes:
 - A node can be a master for some data and slaves for others
 - Nodes are dedicated for master or slave duties

Sharding and Replication on P2P



- Usually each shard is present on three nodes
- A common strategy for column-family databases

Key points

- There are two styles of distributing data:
 - Sharding distributes different data across multiple servers
 - each server acts as the single source for a subset of data.
 - Replication copies data across multiple servers
 - each bit of data can be found in multiple places.
- A system may use either or both techniques.
- Replication comes in two forms:
 - Master-slave replication makes one node the authoritative copy that handles writes while slaves synchronize with the master and may handle reads.
 - Peer-to-peer replication allows writes to any node; the nodes coordinate to synchronize their copies of the data.
- Master-slave replication reduces the chance of update conflicts but peer-to-peer replication avoids loading all writes onto a single point of failure.