CMPUT391

High Performance Databases and “NoSQL”

Instructor: Denilson Barbosa

University of Alberta

October 10, 2023

Slides by D. Barbosa, with suggested corrections and improvements by (in alphabetical order) C. Bins, D. Caminhas, K. Guhzva, Q. Lautischer, E. Macdonald, M. A. Nascimento, K. Newbury, M. Strobl, D. Sunderman, K. Wang, and K. Wong.

What is meant by NoSQL?

Is SQL a bad thing?

The “NoSQL movement” started as frontal opposition to relational database systems in general.

Over time, adopters of the movement realized that abandoning all of the relational technologies meant giving up too much.

Today, NoSQL is often used to mean Not Only SQL. In other words, their proponents advocate using systems that support (at least a subset of) SQL in addition to other access methods and languages.

Outline

High Performance Computing Architectures

Partitioned Table Stores

Consistency and Transaction Processing

Key/Value Stores

Distributed File Systems and Map/Reduce

Relational Operators with Map/Reduce

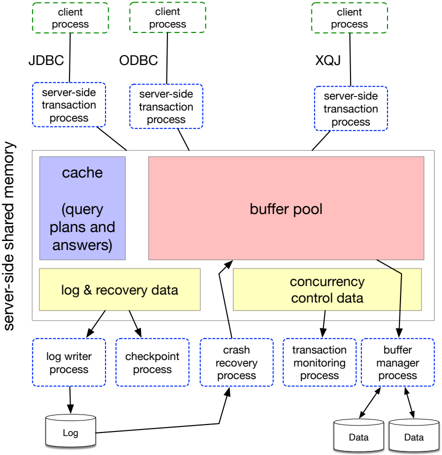
Parting Thoughts

High Performance Computing Architectures

DBMS processes

From the Operating System point of view, a DBMS is implemented by a large number of independent processes, each handling one functionality of the system, and all accessing the same data in memory.

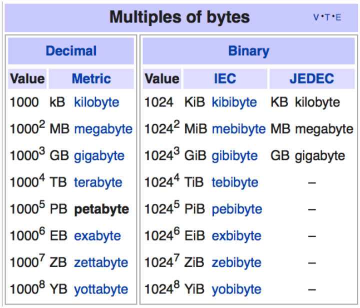
We now look at ways of coping when the number of processes or the volume of data (or both) becomes too high for a “standard” server architecture.



Too much data?

“Large” databases nowadays are measured in **peta**bytes.

Financial institutions and retailers are probably among the ones with the largest relational databases out there.



Multiples of bytes

V·T·E

Decimal			Binary				
Value	Metric		Value	IEC		JEDEC	
1000	kB	kilobyte	1024	KiB	kibibyte	KB	kilobyte
1000^2	MB	megabyte	1024^2	MiB	mebibyte	MB	megabyte
1000^3	GB	gigabyte	1024^3	GiB	gibibyte	GB	gigabyte
1000^4	TB	terabyte	1024^4	TiB	tebibyte	–	
1000^5	PB	**petabyte**	1024^5	PiB	pebibyte	–	
1000^6	EB	exabyte	1024^6	EiB	exbibyte	–	
1000^7	ZB	zettabyte	1024^7	ZiB	zebibyte	–	
1000^8	YB	yottabyte	1024^8	YiB	yobibyte	–	

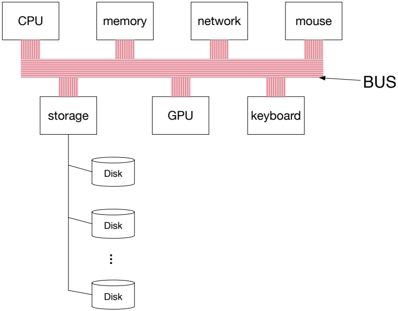
Some known very large databases:

- Facebook reported having a 30PB database in 2011
- Google Maps was reported as 20PB in 2012
- Google’s crawl and associated metadata is reported at 10 exabytes
- Scientific databases are often astronomical (in size too)

With a large database, even the simplest queries become a challenge.

Not too long ago, Google stored 25 billion pages.

At 20KB per page (ignoring figures, etc.), that would be 500 TB of raw data.



How long would it take to run a query to find which web pages contain a link to ualberta.ca on a “standard server”?

With a fast 500MB/s BUS, the table scan would take at least

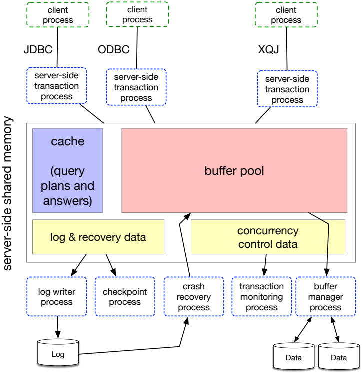
$$\frac{500 \cdot 1024 \cdot 1024 \text{ MB}}{500\text{MB/s}} = 1M\text{seconds} \approx \text{ 12 days}$$

Too many users?

Each connection to the database requires a “server-side” process which takes resources (memory for buffers).

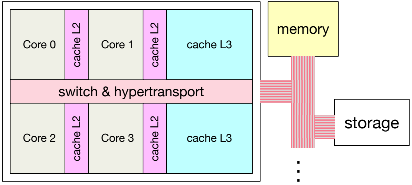
Also, each server-side process competes with the DBMS’s own processes for CPU cycles.

With a single CPU and 1,000 connected users, each requiring 1 second, it will take up to 15 minutes to serve them all.



Architectures for High Performance Computing

The most modest step up from a standard Von Neumann architecture is to use a multi-core[1] CPU:



Pros:

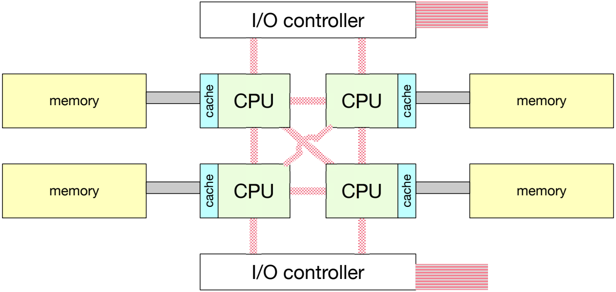
- Real parallelism: as many DBMS processes as the number of cores running at the same time.
- Can move a process from one core to another if needed.

Cons:

- The BUS becomes a bottleneck!

[1] https://en.wikipedia.org/wiki/Multi-core_processor

Modern “single-box” HPC servers have independent (multi-core) CPUs, each with its own memory, interconnected by a fast switch.

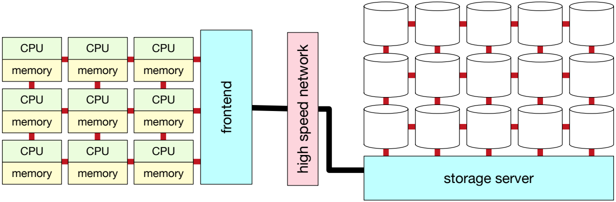


Processes on different CPUs can share memory, but with a Non-Uniform Memory Access (**NUMA**[2]) cost: accessing local memory is cheaper.

[2] `https://en.wikipedia.org/wiki/Non-uniform_memory_access`

Shared Data Architectures

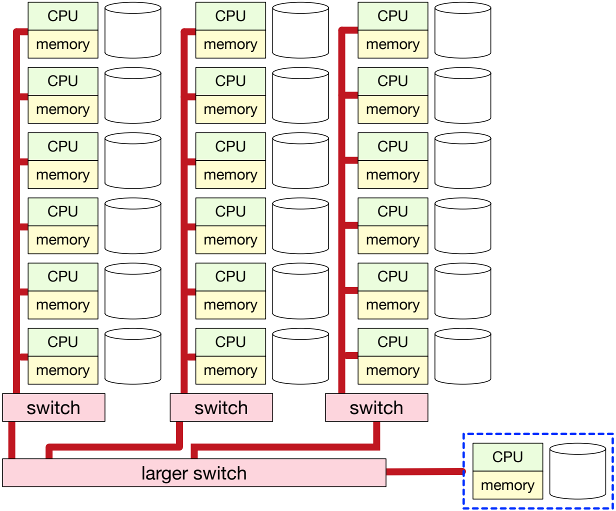
When NUMA is not enough, we move **up** to clusters of computers that shared only the data on disk:



“Single-node DBMS” in this architecture: transaction processes and DBMSs process run on separate nodes and communicate via a local network.

However, transferring data from the storage array becomes a bottleneck. A fast network can reach up to 5GB/s, but that would still mean 1 day to transfer 500 TB each way.

Shared Nothing Architectures



Front-end or Head node

Why is Shared Nothing so Popular?

Shared nothing architectures are made up of inexpensive computing nodes, each with its own disk and memory (typically about $1000 per node).

Shared nothing is ideal for scaling **out** (instead of up): if you need more computing power, buy a few more computing nodes and hook them to the existing cluster.

Shared nothing is great for **embarrassingly parallel** workloads: spread the data among the nodes and run the same query (e.g., finding pages with links to ualberta.ca) on every node at the same time.

Virtualization and cloud computing

Computing clouds are clusters of (shared-nothing) nodes that host a large number of virtual machines (VMs), each with their own OS.

- Multiple VMs run simultaneously on the same physical node.
- VMs can migrate across nodes (like processes migrate across processors).

Cloud Computing can Save (a lot!)

Running an enterprise-scale datacenter incurs many costs:

- Infrastructure: building maintenance, electricity, cooling, ...
- Expertise: good IT staff costs a lot!

If the organization **under**-provisions (i.e., the IT infrastructure is too small), then it loses customers until it grows.

If the organization **over**-provisions (i.e., the IT infrastructure is too big), then it wastes money in maintenance cost.

Clouds $\approx$ Outsourced Infrastructure

Main advantage: let experts run the IT infrastructure and pay only for what you need (or what you use).

Clouds continued

Infrastructure as a Service (IaaS):

- The cloud hosts “bare bones” VMs and the user installs their own OS and software stack

Platform as a Service:

- The cloud offers self-managing VMs that implement a specific service
  - email, word processing, other office applications
  - database services: MySQL, PostgreSQL, ...
  - document storage: MongoDB, CouchDB, ...
  - key/value “data stores”: Bigtable, Cassandra, HBase, ...

Google: `https://cloud.google.com/products/#databases`

Amazon: `https://aws.amazon.com/products/databases/`

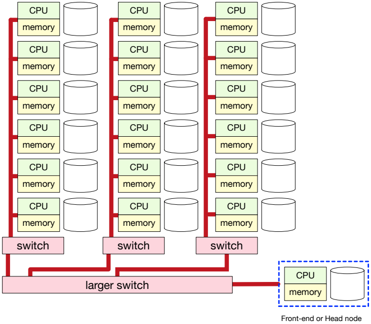
Clouds bring economies of scale

Elasticity

Increasing/decreasing the number of VMs providing any service in response to changes in demand.

Scaling Out

Growth by adding new physical nodes to cope with demand.



Multi-tenancy

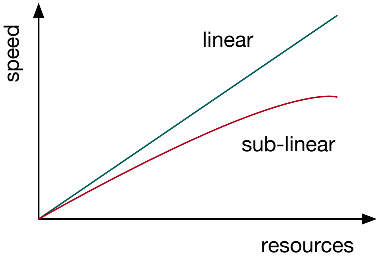
Different clients have their VMs (or database services) running simultaneously on the same physical cluster.

Redundancy is essential

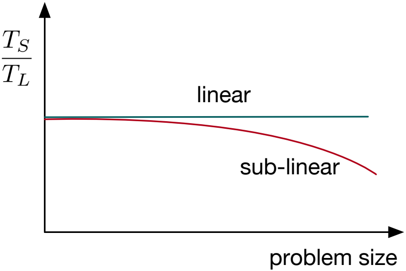
- The cluster components (compute nodes and network connections) are inexpensive and will break at some point
    - Most clusters use commodity hardware;
    - Google, Amazon and other massive IT companies have started to design their own custom compute nodes
- A data center with thousands of nodes will experience several node failures every day
- Challenge #1: restarting the computation after each failure is not an option.... the computation model has to be **fault tolerant**

Redundancy is one way of being fault tolerant: store the same on different compute nodes, across different cluster racks.

Speedup and Scaleup



Speedup



Scaleup

Speedup measures how much faster the system accomplishes a task as more computing resources are added.

Scaleup takes into account the growth of the database. T_S is the time it takes to complete the task on the “small” computer, while T_L is the time on the “large” computer.

HPC trade-offs

A perfect high performance computer (parallel or distributed) would have **linear speedup and scaleup**.

In practice this does not happen. The more components in a system, the more time is needed for them to coordinate amongst themselves.

Example

- A DBMS log writer process responds faster to a server-side transaction process running on the same CUP instead of a process running on a different CPU or a different computer.

Synchronizing independent processes takes time proportional to the complexity of the computer. Also, we have to account for failures and build redundant checks (which also take time).

Synchronization in DBMS workloads

Fact: synchronization time grows with system complexity.

Corollary: tasks that do not require synchronization show better speedup and scaleup.

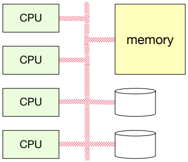
ACID transactions require a lot of synchronization

- Atomicity requires logging.
- Isolation requires transaction monitoring, locking, etc. which must be performed by a single DBMS process.

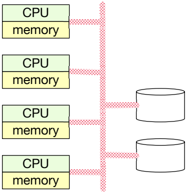
A defining "feature" of many NoSQL systems that scale well is to not provide ACID transactions.

Parallel vs Distributed Database

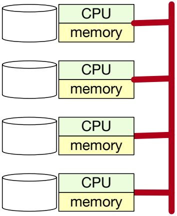
In a single-node parallel database all the data and all DBMS processes reside in the same HPC computer, independently of architecture:



shared memory



shared disk



shared nothing

A distributed database is one made of autonomous "single-node" databases. Typically, each database is located on a different data center (i.e., the nodes are geographically dispersed).

We fill focus on shared-nothing clusters, as they are becoming the most popular (due to the cost benefits of cloud computing).

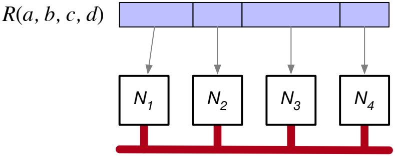
Main questions about HPC NoSQL in the remaining of the notes:

- What data management problems are shared nothing architectures best for?
- What compromises do we make to handle more data or clients relative to what a single-node relational system offers?

Partitioned Table Stores

Data Partitioning

Partitioning a table $R(a, b, c, d)$ within a cluster, can help *balance* the storage and query/update workload across nodes.



- Based on ranges of one or more partition attributes.

  Example, partition on $R(a)$: node$(t) = \begin{cases} N_1 & t.a < k_1 \\ N_2 & k_1 \leq t.a < k_2 \\ \ldots \end{cases}$

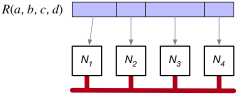
- Based on hashing of one or more partition attributes.

  node$(t) = h(t.a) \bmod 4$

- In a round-robin fashion (i.e., tuples go to nodes based on the order they are inserted).

Cost of Finding a Tuple

Recall that the costs of querying, deleting and updating a tuple all depended on the cost of finding a tuple in te first place.



Cost of search based on $R(a) = v_1$.

(1) Finding the node with the tuple:

- If a is the partition attribute (range partition or hashing), we can quickly find the single node where the tuple is.
- With round-robin partition **or** if a is not the partition attribute, we need to search for the tuple in every node.

(2) Performing the search in each node: $O(|R|)$ or $O(\log_k |R|)$ depending on whether there is an index on a.

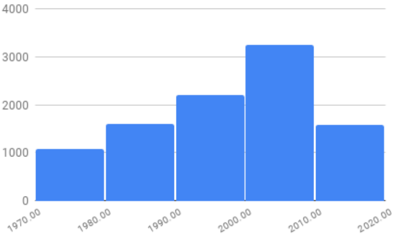
Data Distribution

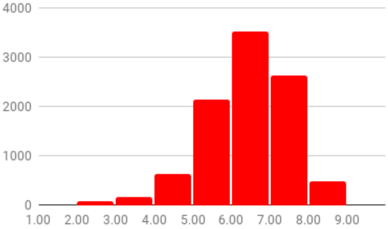
How many tuples will there be in each partition of R into $R_1 \cup R_2 \cdots \cup R_n$

- With **round-robin**: $T(R_i) = \frac{1}{N}$.
- With **range partitioning** and with **hash partitioning**, it depends on how the distribution of the values of the partition attribute(s).
  - If the values of $R(a)$ are uniformly distributed and $V(R, a) > N$, then $T(R_i) = \frac{1}{N}$.
  - But what if they are not?

If the values of $R(a)$ are not uniformly distributed then hashing and range partitioning will assign (possibly a lot) more tuples to a few nodes, causing the load in the cluster to be **unbalanced**.

Ex: suppose we partition Movie(title, year, imdb, director) by year (left) or imdb score (right):





Load imbalance is a problem

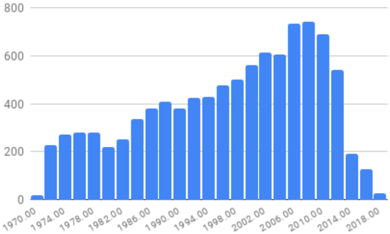
Nodes with more data perform **a lot more** work over time.

Virtual nodes

One way to combat load imbalance is to add a level of indirection by using more *virtual nodes* than real nodes:

- Use any partition scheme to assign tuples to virtual nodes.
- Assign virtual nodes to real nodes in a round-robin fashion[3].

In effect, using *virtual* nodes is the same as increasing the granularity of the data partitioning strategy.



Randomizing the assignment of virtual nodes to physical nodes balances out the workload of the physical nodes.

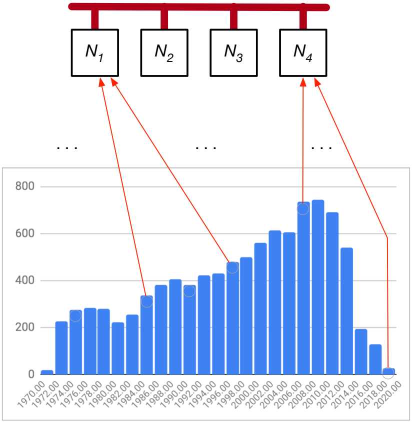
[3] The other two techniques would work as well.

Each virtual node corresponds to a narrower slice of the data.

Each physical node is assigned multiple slices, in a way that the total load in each physical node is as egalitarian as possible.

Ideally, each physical node should have

$$T(R)/N \text{ tuples.}$$



Elasticity

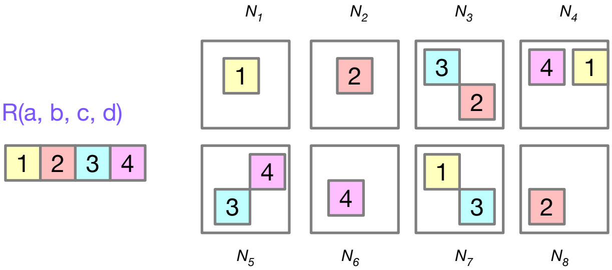
The use of virtual nodes to partition the data is consistent with the “elastic nature” of cloud computing:

- When the cluster grows (i.e., new physical nodes are added), existing virtual nodes can be re-assigned and migrated.
- Also, each “slice” of the data can be further divided into even narrower slices to help re-balance the load among physical nodes.

Replication, Redundancy and Availability

Nodes in a cluster are expected to fail often and without warning.

A good way to **avoid data loss** is to keep multiple copies of the data, so that even the failure of a few nodes does not mean the entire table is lost.

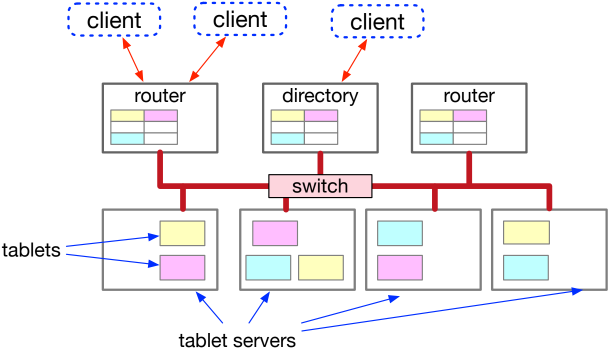


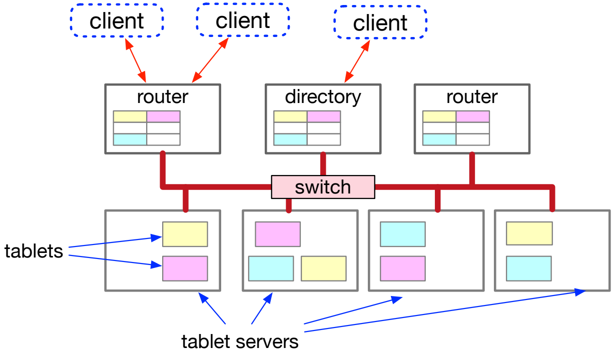
This also helps with query answering: always send request to read data to the nodes with least load.

Prototypical “Table Store”

The table is partitioned into tablets, distributed and replicated across tablet servers. A directory process, running in the **primary node** of the cluster, keeps a mapping between tablets to nodes.

To increase availability and redundancy, directory process is replicated in multiple nodes, each running a data router process.





Small deployments will typically have a small number of physical nodes, each running multiple processes.

- Routing process: redirect requests based on partition attribute.
- Data storage process: read/write operations on tuples.
- Load re-balancing process.
- Query/transaction execution process!

Updating the data

Inserting new tuples requires choosing a primary tablet to hold the tuple and updating the directory accordingly.

To delete a tuple or to modify a non-partition attribute of a tuple, we:

(1) Find the tablet with the **primary** copy of the tuple.
(2) Perform the update/deletion.
(3) Synchronize all replicas of that partition.

Updating the partition attribute(s) is often implemented as deleting the tuple followed by an insertion of the “modified” one.

Synchronization

Recall that in these systems data loss is prevented by having each tuple stored in multiple places.

Which means we need to replicate every insertion and every update in the primary tablet to its replicas.

How do we ensure the replicas are synchronized?

- Two-phase Commit (2PC) protocol[4] ensures synchronous and atomic updates across replicas.
- Persistent-messaging implements a more relaxed, **eventual**, consistency model.

[4] https://en.wikipedia.org/wiki/Two-phase_commit_protocol

Why do we need synchronization again?

Synchronization is necessary (in single-node or multi-node systems) to ensure that transactions are atomic and durable and to ensure that the transaction schedule is conflict-serializable.

In other words, synchronization is needed if we want ACID transactions.

Not all applications need synchronization though...

As we will discuss (soon) some applications can tolerate data inconsistencies as long as they can be (eventually) detected and corrected.

Trade-off: give up consistency to gain in speed.

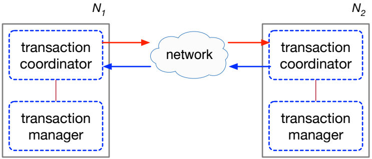
Consistency and Transaction Processing

Distributed Transactions

Nodes in the cluster execute two kinds of transactions:

- **Local transactions** only read/write data on the node running the transaction.
- **Global transactions** read/write data on multiple nodes.

The **transaction coordinator** synchronizes the individual (local) transactions running on different nodes via the network.



The **transaction manager** is responsible for logging and concurrency control only within the node where it runs.

Two-Phase Commit Protocol (2PC) without failures

2PC ensures transactions commit **if and only** all replicas are consistent. Assume node N_1 (with transaction coordinator TC_1) starts transaction T_x.

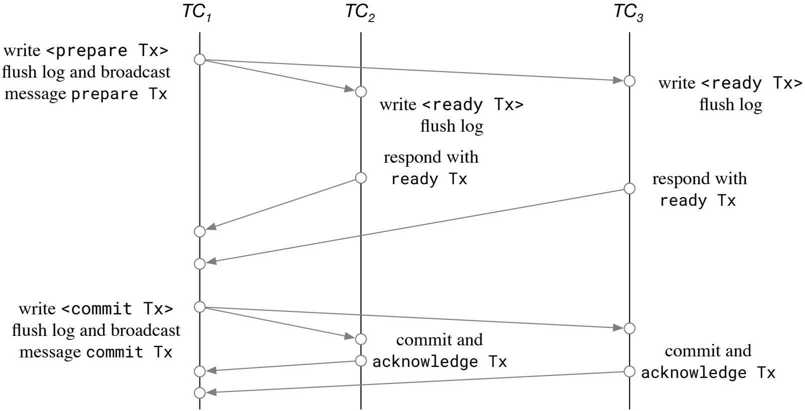
(1) TC_1 periodically sends messages to all other TC_i that are affected by the transaction, e.g., so that they write the new values to their logs, etc.

(2) **Phase 1**: starts when the transaction is ready to commit; TC_1 writes `<prepare Tx>` to its log, flushes its log, and sends a **prepare** message to all other TC_i involved.

All replicas write `<ready Tx>` to their log and respond to TC_1 that they are ready.

(3) **Phase 2**: starts when TC_1 gets the ok from all replicas, it writes `<commit Tx>` to its log, flushes its log, and sends a commit message to all other TC_i.

2PC timeline



TC_2 and TC_3 are the replicas of TC_1

2PC: correctness

Rule #1: TC_1 will only start the attempt to commit if all replicas respond with a `<ready Tx>` message.

If a replica cannot commit or does not respond after some time limit, the transaction is aborted and all replicas are notified to abort.

Rule #2: TC_1 knows all replicas are synchronized once they all respond with a `<acknowledge Tx>` message.

The protocol deals with different kinds of failures both in the coordinator node or in the replica node.

2PC: replica failure

If the coordinator TC_1 detects that replica node N_j failed **before** responding with `<ready Tx>`, it assumes the replica is not ready and aborts the transaction everywhere, and no inconsistencies arise.

If the coordinator TC_1 detects that replica node N_j failed **after** responding with `<ready Tx>`, it continues with the protocol with the other replicas, ignoring the node failure.

Once the failed replica N_j is back up, it performs a crash recovery algorithm (next slide).

2PC: crash recovery — replica

Upon restart, replica N_j checks its log to undo/redo transactions almost as in the single-node case (recall undo/redo logging from before). For each transaction T_x in the log:

- If `<commit Tx>` is found in the log, **redo** T_x.
- If `<abort Tx>` is found in the log, **undo** T_x.
- If only `<ready Tx>` is found in the log, then the replica must consult the transaction coordinator TC_1 to figure out the if the transaction committed or aborted.
  - If the coordinator is unreachable, N_j queries all other replicas to check the status of the transaction.
  - Until then, N_j **neither** commits nor aborts.
- If not even `<ready Tx>` is in the log, the replica did not respond (and the transaction has been aborted by the coordinator), so the replica aborts the transaction.

2PC: coordinator failure

If the coordinator fails or becomes unavailable during the execution of the transaction, the replicas continue executing the transaction to the extent possible:

- If at least one replica wrote `<commit Tx>` to its log, then all active replicas proceed to commit the transaction.
- If an active replica wrote `<abort Tx>`, then all active replicas must abort.
- If at least one replica does not have `<ready Tx>` in its log, the coordinator cannot have started the commit phase. In this case, the replicas assume the coordinator decided to abort (based on timeout) and they all abort.
- If all replicas have only the `<ready Tx>` in their log, then the replicas must wait for the transaction coordinator TC_1 to figure out the if the transaction committed or aborted.

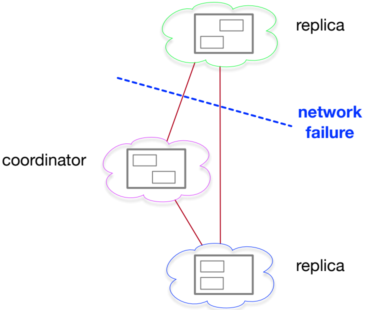
2PC: crash recovery — coordinator

The crash recovery for the coordinator is identical to that of a replica N_j:

- Redo and commit transactions marked `<commit Tx>` the log.
- Undo and abort transactions marked `<abort Tx>` in the log.
- Consult the replicas about transactions marked `<ready Tx>` in the log.
- If not even `<ready Tx>` is in the log, the replicas did not respond before the crash, and everyone decided to abort the transaction.

Network partition and delayed transactions

A severe network failure might partition the system into two disconnected subsystems.

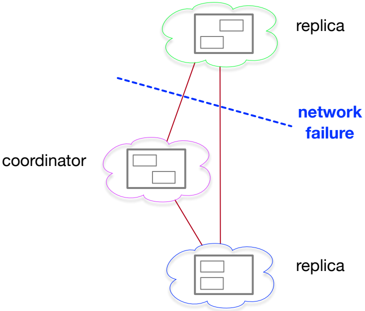


The 2PC protocol dictates how the coordinator and the replicas behave if the failure happens during the execution of a transaction.

However, can the same query have a different answer depending on the node it is issued?

Also, should new transactions be accepted while the network is partitioned?

Example: if the isolated replica could not decide whether or not to commit or abort a transaction T_x, it **cannot** accept new transactions that use data written by T_x!



But what about the other nodes in the “other half” of the system?

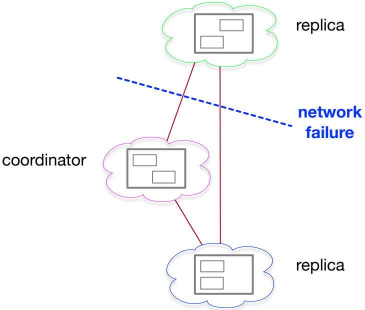
What if the network failure may last for a long time?

- Should the system stop accepting transactions?
- What about queries?

Accepting Transactions in a Partitioned System

The blocking problem

A node isolated from the rest of the network it may not know if a transaction committed or aborted, preventing it from accepting new transactions (e.g., to avoid uncommitted reads).



But distributing the data was meant to bring **high availability**. Instead, the whole system becomes unavailable when there is a failure somewhere else.

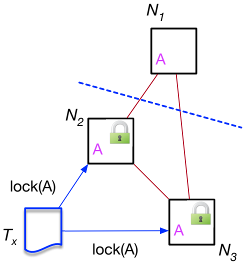
High Availability and Data Consistency are conflicting goals...

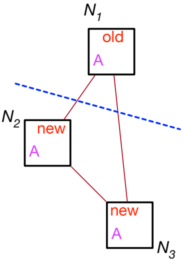
Eventual Consistency With Majority Locking

One way to allow the system to tolerate node or network failures is to allow a transaction T_1 to write a database element A if it can acquire locks on the (simple) majority of the nodes with copies of A.

Timestamps (agreed upon at locking time) are used to record the version of A that is written.

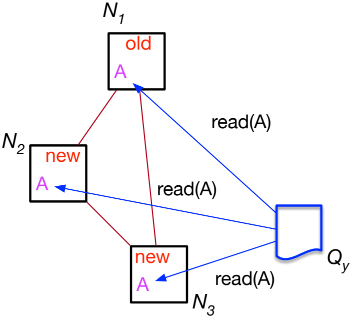
After the update commits the system becomes inconsistent: different nodes have different "most recent" values of the same element.





Inconsistent data can be detected in many ways:

- A node with a copy of A re-joins the network after a failure and asks its peers for missed transactions.
- Another node requests A from all copies and gets inconsistent answers.



In summary, majority voting ensures the system remains available during the failure of a node, but incurs too many reads to work well.

The CAP “Theorem”

Although this is not a formal statement, it is generally accepted that no distributed database system that enforces ACID transactions can achieve these three properties **simultaneously**:

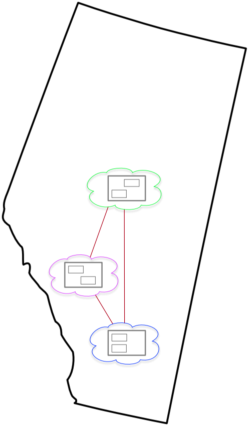
- **C**onsistency: all replicas have the same value.
- **A**vailability: the system accepts transactions at any time.
- **P**artition tolerance: the system operates normally even under a network partition failure.

Instead of ACID, NoSQL systems are BASE:

- **BA**sically available: it works as long as the majority of the replicas are up.
- **S**oft state: inconsistencies are time-stamped.
- **E**ventually consistent: replicas do synchronize after a while.

Synchronization across data centers

Many applications require the data to be **distributed** across geographically dispersed data centers to reduce the chance of critical data loss due to catastrophic failures in one location (e.g., fire in one data center).



The 2PC protocol works on these systems as well, although network latency and connectivity can introduce significant delays in replica synchronization.

It is up to the database administrator to evaluate the trade-offs between responsiveness and data consistency.

Moral of the story?

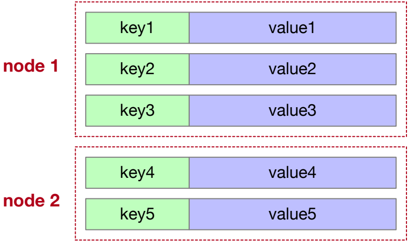
If an application cannot tolerate inconsistent data (e.g., a banking application), then it will use the 2PC protocol and require all locks to be available before writing.

Most distributed applications (e.g., social networks) can tolerate inconsistent data. The worst that can happen is that users connected to different nodes (e.g., opposite coasts of Canada) of the system don’t see each others posts for some time.

Key/Value Stores

Key/Value Stores

Key/Value stores have become popular recently. Example systems include Apache HBASE, Amazon's DynamoDB and Google's BigTable.



node 1	key1	value1
	key2	value2
	key3	value3
node 2	key4	value4
	key5	value5

Fundamentally, these systems implement partitioned tables as discussed before with the added restriction that the partition is defined on a singleton key attribute.

Why are Key/Value Stores So Popular?

Except for the keys, each tuple can have its own “schema”, with its own columns. Also, the “columns” can store non-1NF data, like arrays and lists or other data structures.

In other words, these key/value stores are good for storing **semistructured data**!

Example:

	isbn	title	author	year	follows
node 1	0-262-03141-8	Introduction to Algorithms	[“Cormen”, “Leiserson”, “Rivest”]	1990	
	0-521-89560-X	Causality	“Pearl”	2009	
node 2	0-262-03293-7	Introduction to Algorithms	[“Cormen”, “Leiserson”, “Rivest”, “Stein”]	2001	0-262-03141-8

	isbn	title	author	year	follows
node 1	0-262-03141-8	Introduction to Algorithms	["Cormen", "Leiserson", "Rivest"]	1990	
	0-521-89560-X	Causality	"Pearl"	2009	
node 2	0-262-03293-7	Introduction to Algorithms	["Cormen", "Leiserson", "Rivest", "Stein"]	2001	0-262-03141-8

Support for application-level constraints?

- ex: every ISBN in a "follows" attribute must match an ISBN of some other "row" in the table.

The programmer is on their own here… in general, key/value stores **do not support** constraints like foreign keys. In fact, they don't even support triggers easily.

Many “document stores” are just key-value stores under the hood, where the **keys are system-generated** at insertion time.

These systems allow programs to Create, Read, Update, or Delete objects (CRUD).

Systems like Elasticsearch also automatically populate indexes as documents are loaded to the “table”. For example, Elasticsearch indexes allow it to very quickly find documents that match phrases or keywords provided by the user or an application, even for very large collections.

CMPUT361 covers those kinds of indexes and a lot more about document processing.

Example: storing document into an Elasticsearch “table” books:

```
POST books/_doc/
{
    "isbn" : "0-262-03141-8",
    "title" : "Introduction to Algorithms",
    "year" : 1990,
    "authors" :  [ "Cormen", "Leiserson", "Rivest"]
}
```

Sample output:

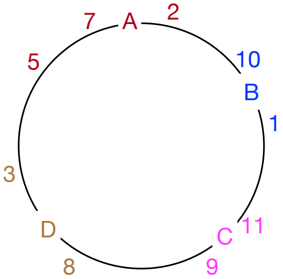
```
{
    "_shards" : { "total" : 2, "failed" : 0, "successful" : 2 },
    "_index" : "books",
    "_type" : "_doc",
    -:"_id" : "W0tpsmIBdwcYyG50zbta":-,
    "_version" : 1,
    "_seq_no" : 0,
    "_primary_term" : 1,
    "result": "created"
}
```

Consistent Hashing

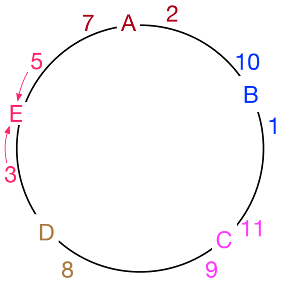
Consistent hashing is a key ingredient in distributed key/value stores that allows for load balancing even when nodes join or leave the system.

Nodes (A, B, C, and D) and objects (1, 2, 3, ...) are hashed to the same circular space (e.g., 32bit hash values with wraparound).

Objects are assigned to the **closest** available node in the system.



When a new node joins the cluster, it broadcasts its hash value to other nodes, who respond with a list of documents that are closest to the newcomer.



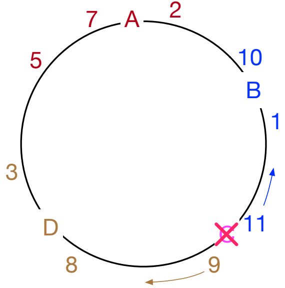
Ex: D will take on documents 3 and 5.

Documents migrate over time.

If a request for a document that has not migrated yet arrives, the new node can relay that to the old node that still has the data.

When a node announces it intends to leave the cluster, it sends lists of documents that are to be migrated to its closest neighbors.

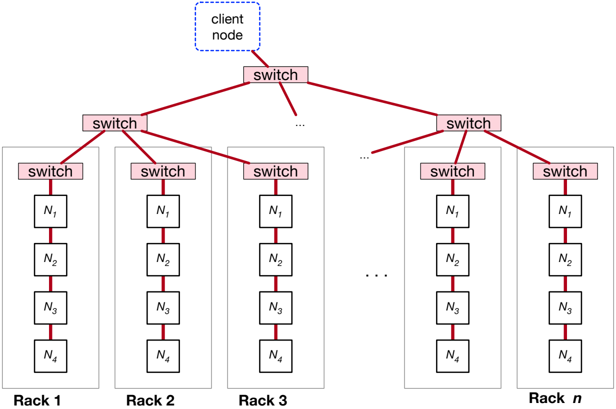
Again, documents migrate gracefully and over time.



If a node fails, its neighbors consult all replicas of that node, to take over the documents that are now (temporarily) unavailable.

Distributed File Systems and Map/Reduce

Data Processing with Map/Reduce



We now look into using a shared-nothing cluster as a parallel computer for data processing.

Map/Reduce is a functional programming paradigm that can be easily parallelized.

Very popular inside Google from 2004: `https://research.google.com/archive/mapreduce-osdi04-slides/index.html`

Facebook and Yahoo both invested in the open source Apache Hadoop project

`https://wiki.apache.org/hadoop/PoweredBy#F`

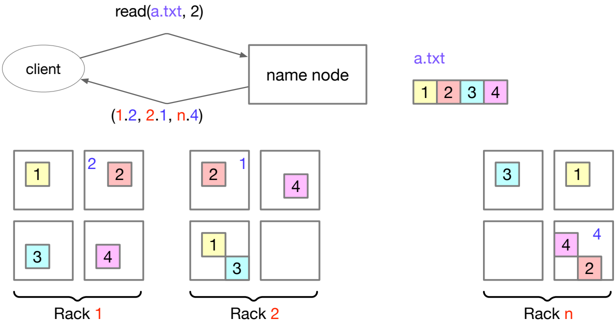
`https://wiki.apache.org/hadoop/PoweredBy#Y`

HDFS: Redundancy and Load Balancing

Each blocks of a file is replicated (usually twice) across nodes and across racks (to account for network component failures).

The “directory” is kept by a “name node” (in Hadoop terminology), which tells the client which nodes have the data.

Any node with the data can process it.



Programming Parallel Computers is Hard

Challenge #1: it is hard to design the control flow when multiple computations happen *at the same time* and in different computing nodes...

- It is **very hard** to debug such designs.

Challenge #2: giving the programmers a programming paradigm simple enough so that we can easily express computations, yet **powerful enough** to be practical.

MapReduce is a simple functional and powerful programming paradigm. Computations in MapRedure are:

- easy to understand by humans
- can be executed in parallel in massive clusters

Data locality is important

The goal of using a cluster was to avoid the communication cost of moving all the data around.

- Some communication is unavoidable, however.

Parallel programs often alternate between:

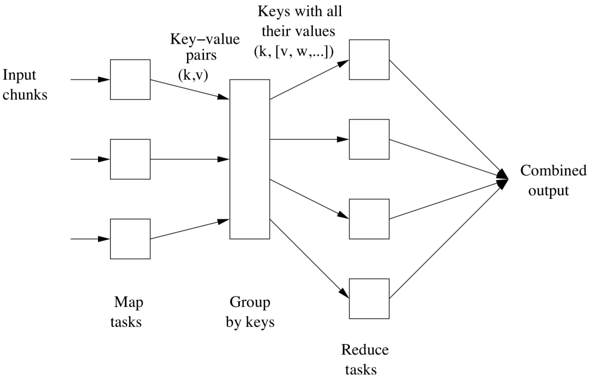
- Computing with *local data* (no communication).
- Exchanging results of local computation with other nodes.

Challenge #3: avoiding “all-to-all” communication model where every node exchanges data with every other node

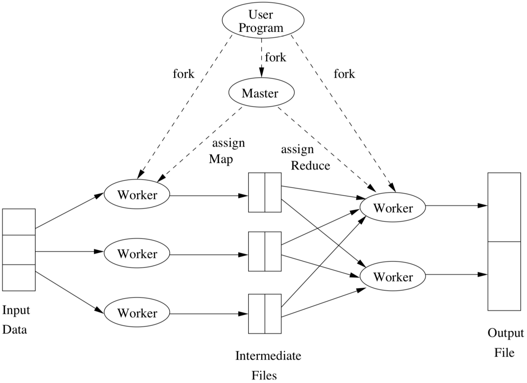
MapReduce computations work in stages: map operations work on local data, reduce operations aggregate results. Communications are directed.

Stages of MapReduce computations

- map operations work on local data, and produce *key*, *value* tuples
- tuples are grouped together by key
- all tuples with the same key are sent to the same reduce task



Execution of a MapReduce job



The program specifies the `map()` and `reduce()` functions

The Master process starts “worker” processes to complete all `map()` tasks; once they are done, the Master starts “workers” for the `reduce()` tasks.

Word count with MapReduce

Suppose we want to count the number of times each word appears in a large corpus of texts (e.g., millions of documents).

- Each compute node has many documents in its local storage
- No compute node has all occurrences of any single word

General idea:

(1) Map: emit tuples $(w, 1)$ for every occurrence of a word in a document

(2) Group all streams of tuples with the same word

(3) Reduce: return tuple (w, l) where l is the number of tuples for word w

Map operation

For every file *f* stored locally, do:

- normalize and tokenize *f* into a list of words $[\texttt{w}_1, \texttt{w}_2, \ldots, \texttt{w}_n]$
- for every word *w* in $[\ \texttt{w}_1, \texttt{w}_2, \ldots, \texttt{w}_n]$
  - $\rightarrow$ emit tuple `(w, 1)`

Reduce operation

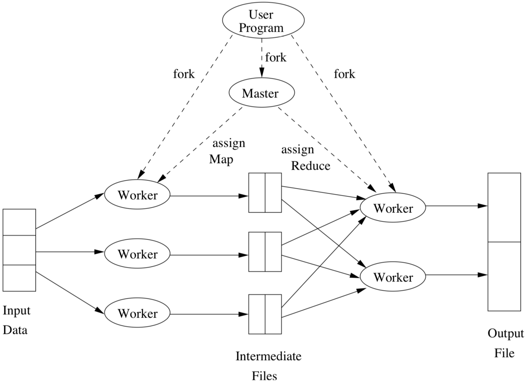
Input: stream of tuples $t_1 = (w_j, 1), t_2 = (w_j, 1), \ldots, t_k = (w_j, 1)$

Output: k

Where does the output go?

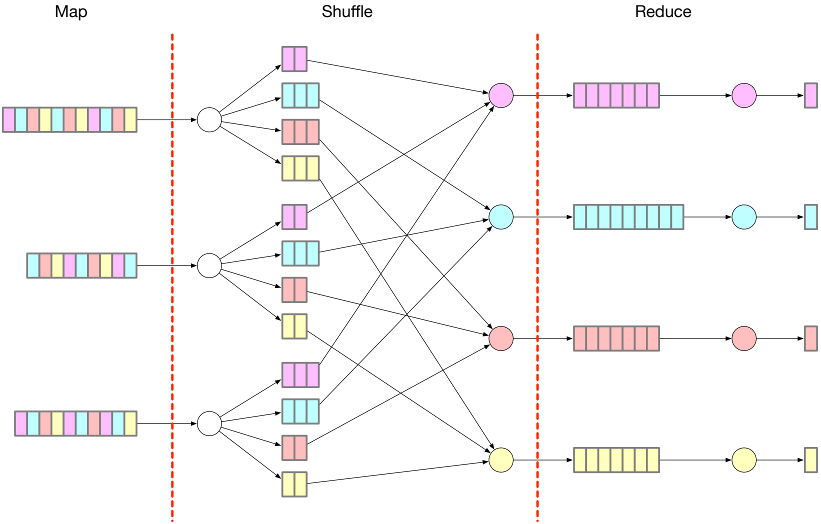
The reduce operation can write data to a file. So, each compute node could open a file for each word it received...

Execution of a MapReduce job

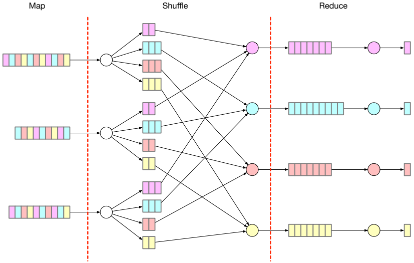


The program specifies the `map()` and `reduce()` functions

The Master process starts “worker” processes to complete all `map()` tasks; once they are done, the Master starts “workers” for the `reduce()` tasks.


Map
Shuffle
Reduce

Networking Cost



The master assigns keys to nodes, deciding which nodes will perform which `reduce()` operations.

Then, the nodes “shuffle” the stream of tuples, sending each tuple to the right `reduce()` node.

NOTE THAT each node keeps at least one list of tuples though!

Obvious optimization

The example so far is the “textbook” word count, where each tuple has count 1 for every word.

An obvious optimization would be for the `map()` operation to emit one tuple per word among all documents it reads. This is possible, of course, only if the computing node has enough RAM to keep that map in memory.

But even if the node does not have a lot of memory, it can keep accumulating word counts until all RAM is used up. At that point, the node can emit all tuples and start counting again.

Multi-stage computations

It is possible to have more complex computations involving several MapReduce phases. Below is one example MapReduce computation following finding individual word counts.

Note that each node is left with a list of tuples, each with the count of an individual word, and that no two nodes have the count of the same word.

Finding total number of words:

`map()`

Go over all words in the node and emit a single tuple (*k*,*total*) where total is the sum of individual word counts in the node.

`reduce()` **– note that all tuples have the same key *k***

Sum up the values in the incoming tuples.

Finding word(s) with highest frequency:

`map()`

Go over all word counts in the node; emit a single tuple $(k,(c,(w_1, w_2,\ldots, w_n)))$ where:

- k is any constant
- c is the highest word frequency among all words in the node
- $w_1, w_2,\ldots, w_n$ are all the words with count c

`reduce()` – note that all tuples have the same key *k*

Go over the stream of tuples; find the highest count; return all words with that count.

Note that the highest count can be found in multiple nodes… so the reducer may have to merge word lists coming from different mappers.

Matrix Operations in Map/Reduce

Google's PageRank algorithm can be elegantly stated and efficiently computed as a series of matrix multiplications, which they implemented using Map Reduce for scalability reasons.

In fact, Google was probably among the first company to build a fortune on linear algebra.

Let A be a $m \times n$ matrix (e.g., the adjacency matrix of the Web graph), and $\vec{v}$ be an n-dimensional vector.

Vector-matrix product $x_i = \sum_{j=1}^{n} a_{ij} v_j$

- `map()`: taking the entire $\vec{v}$ as input, compute $a_{ij} v_j$ for all cells of the matrix stored at the node; emit tuple $(i, m_{ij} v_j)$.
- `reduce()`: sum all partial values of $a_{ij} v_j$ for each key i_x, and write i_x and the sum to the output.

Let A be a $m \times p$ matrix B be an $p \times n$ matrix.

The product AB is the $m \times n$ matrix C, in which element $c_{ij} = \sum_k a_{ik} b_{kj}$ and

One way to compute C is through two map/reduce stages:

Matrix Multiplication Stage 1

- `map()`: emit each matrix element for whatever matrix data is in the node. That is, emit all $(k, (A, i, a_{ik}))$ and $(k, (B, j, b_{kj}))$.
- `reduce()`: for each pair (A, i, a_{ik}) and (B, j, b_{kj}) that agree on the same k, write the tuple $(k, i, j, a_{ik} b_{kj})$.

At the end Stage 1, we can build a separate list for each value of k: $(i_1, j_1, v_1), (i_2, j_2, v_2), \ldots, ((i_p, j_p), v_p)$ with all combinations of i, j and factors whose sum that will go into cell c_{ij}.

The first stage computes, effectively lists of the form

$$(k, [(i_1, j_1, v_1), (i_2, j_2, v_2), \ldots, ((i_p, j_p), v_p)]$$

In this stage we compute an *aggregate* (the sum) of these lists.

Matrix Multiplication Stage 2

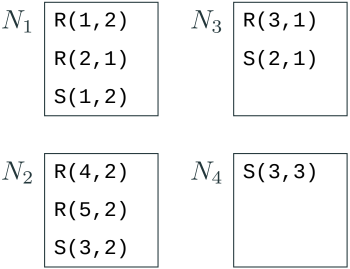
- `map()`: go through the tuples from the previous stage and emit $((i_1, j_1), v_1), ((i_1, j_1), v_2), \ldots ((i_p, j_p), v_p)$.
- `reduce()`: sum up all values with the same (i, j), and write $((i, j) \sum_p v_p)$ to the final output.

Relational Operators with Map/Reduce

Relational Operators with Map Reduce

All basic relational operators can be easily implemented with Map/Reduce reading from a relational store in each cluster node[5].

Example dataset, with schema $R(a, b), \;\; S(c, d)$.



N_1
R(1,2)
R(2,1)
S(1,2)

N_3
R(3,1)
S(2,1)

N_2
R(4,2)
R(5,2)
S(3,2)

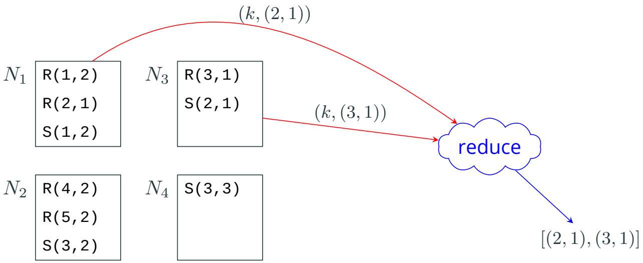
N_4
S(3,3)

[5] Apache's Pig `https://pig.apache.org` is a fairly powerful SQL-on-map-reduce.

Selection $\sigma_C R$ with a single reducer.

- `map()`: go through every tuple in the node, emit each tuple satisfying the selection with reduce key k.
- `reduce()`: collect all tuples.

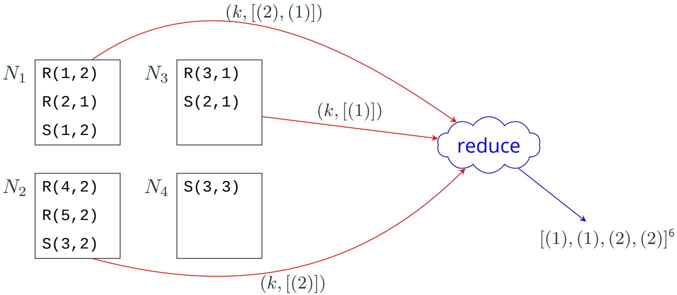
Example: $\sigma_{b=1} R$



Projection $\pi_{a_i,\ldots,a_j} R$ with a single reducer.

- `map()`: go through every tuple in the node; collect all projected tuples; emit tuple $(k, [(v_1), (v_2), \ldots, (v_x)])$ with unique projected tuples.
- `reduce()`: gather all tuple streams!.

Example: $\pi_b R$



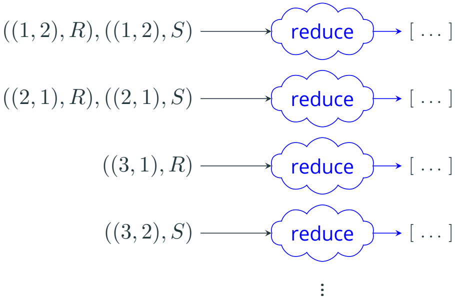
[6] Recall that the SQL version of project does not remove duplicates.

Set Operations

General idea: every mapper emits each original tuple as the (reduce) key and the relation name as the value.

In our example, this would produce the streams to the right.

The reducer implements the set operator.



$((1,2),R),((1,2),S) \longrightarrow$ reduce $\longrightarrow [\,\dots\,]$

$((2,1),R),((2,1),S) \longrightarrow$ reduce $\longrightarrow [\,\dots\,]$

$((3,1),R) \longrightarrow$ reduce $\longrightarrow [\,\dots\,]$

$((3,2),S) \longrightarrow$ reduce $\longrightarrow [\,\dots\,]$

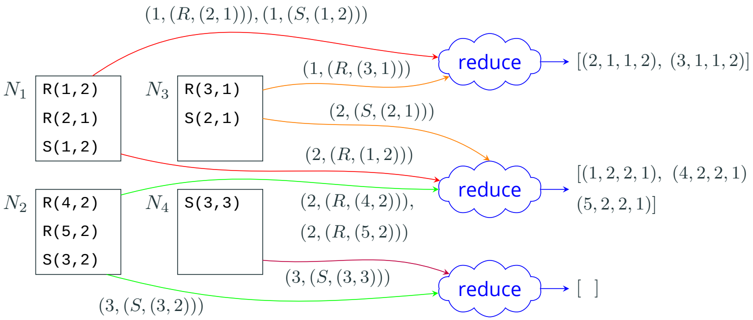
$\vdots$

- $R \cap S$: reducer keeps keys with both values.
- $R \cup S$: reducer keeps all unique keys.
- $R - S$: reducer keeps keys with value R but not S.

Join $R \bowtie_{x=y} S$

- `map()`: emit tuples from R with attribute $R.x$ as key, and tuples from S with $S.y$ as key.
- `reduce()`: iterate through the incoming lists with matching values!

Example: $R \bowtie_{b=c} S$



N_1

R(1,2)
R(2,1)
S(1,2)

N_2

R(4,2)
R(5,2)
S(3,2)

N_3

R(3,1)
S(2,1)

N_4

S(3,3)

$(1,(R,(2,1))),(1,(S,(1,2)))$

$(1,(R,(3,1)))$

$(2,(S,(2,1)))$

$(2,(R,(1,2)))$

$(2,(R,(4,2))),$

$(2,(R,(5,2)))$

$(3,(S,(3,3)))$

$(3,(S,(3,2)))$

reduce $\rightarrow$ $[(2,1,1,2),\ (3,1,1,2)]$

reduce $\rightarrow$ $[(1,2,2,1),\ (4,2,2,1)$ $(5,2,2,1)]$

reduce $\rightarrow$ $[\ \ ]$

SQL in Map/Reduce

Except for fixpoint-semantics recursion, pretty much all of SQL can be easily computed with Map/Reduce:

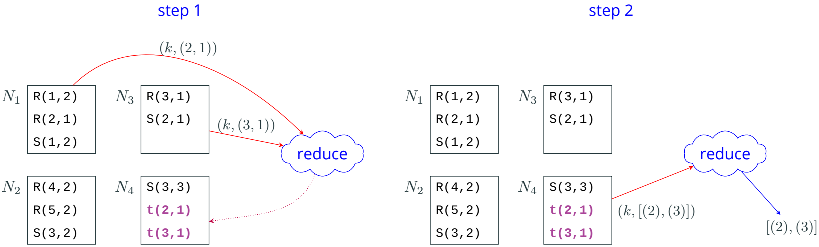
- The “bag” versions of the set operations is also easily done in Map/Reduce: all that is needed is to count the number of (reduce) keys with value R or S in each stream.
- Aggregations (`GROUP BY`) and set functions are also easily computable with Map/Reduce.
- Duplicate elimination is also trivial when the reduce key is a tuple.

Multi-operator Queries

We answer a query like $\pi_{a_1,\ldots,a_n}(\sigma_C(R))$ in two map/reduce processes:

(1) Materialize t $\leftarrow \sigma_C(R)$ in a temporary table.

(2) Compute $\pi_{a_1,\ldots,a_n}(\mathtt{t})$ over the temporary table.



How many reducers to use?

Recall that in every map/reduce computation:

- The number of `map()` tasks is the same as the number of computing nodes.
- The number of `reduce()` tasks is the same **as the number** of reduce keys!

Why does this matter?

If the first step uses a single reducer:

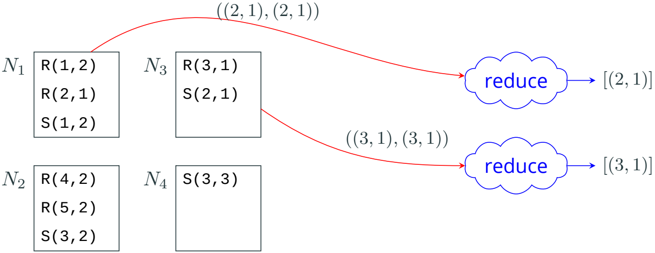
- A single node will store all tuples from the first step[7].
- Only one `map()` task will run in the second step.

[7] Of course, the cluster may be setup to replicate the data across multiple nodes, alleviating the problem.

Selection $\sigma_C R$ with multiple reducers.

- `map()`: go through every tuple in the node, emit each tuple satisfying the selection using the tuple itself[8] as key.
- `reduce()`: collect all copies of each tuple.

Example: $\sigma_{b=1} R$



[8] A hash of the tuple would also work, and save on communication costs.

How many reducers should one use?

- A single reducer can intuitively act as the “server-side” transaction process for each request by a connected user/application.
- With multiple reducers (which run as independent processes), one can spread the tuples in the “temporary table” across more nodes, increasing the number of `map()` tasks that **run in parallel** in subsequent steps.
- Thus, multiple reducers seem better for intermediate query operations, while the single reducer approach seems better for the root node of the query.

Map/Reduce in NoSQL systems

Some document stores like CouchDB use Map/Reduce for parallel query processing [9].

For example, one can use `map()` functions to select multiple (fragments) of JSON documents in the cluster that satisfy a predicate, querying all documents in the cluster simultaneously, and the `reduce()` function to compute an aggregate answer.

```
function map(doc) {
  if(doc.author.includes("Manning")) {
      emit(doc.publisher,
       doc.title);
  }
}
```

```
function reduce(key, values) {
  var cnt = 0;
    for(var idx in values) {
      cnt = cnt + 1;
  }
  return (key, cnt);
}
```

[9] https://docs.couchdb.org/en/stable/ddocs/views/intro.html

Parting Thoughts

High Performance Computing architectures, in particular, shared-nothing clusters of computers organized into a “cloud”, are becoming the norm in supporting database applications.

Two key advantages of this approach are distributing the data across nodes to reduce the risk of data loss and increase availability of the data.

The compromise is that updates to the data require a lot more effort: multiple nodes need to coordinate via the network, which is much slower.

As a result, most NoSQL systems do not support complex constraints and do not offer full SQL support.