DDIA notes

Sunday, April 18, 2021 5:39 PM

Chapter 3 Storage and Retrieval

1. There are 3 dimensions to DB performance: Read(R) efficiency, Update(U) efficiency ar
   there are OLAP, OLTP and memory optimized DBs http://daslab.seas.harvard.edu/rum
2. Ways of making indexes
   a. Hash indices (Log Structured)
      i. Use cases: In-memory DB
         1. {keys hashes -> disk location} are in RAM
         2. values are stored on disk
         3. Good for things like counters where key size is limited but frequent u
         4. Not RDBMS
         5. Eg: Bitcask (default in Riak)
      ii. Pros:
         1. quick lookup of keys
         2. High write throughput
      iii. Cons:
         1. number of keys in DB are limited by RAM available
         2. Since we hashed the key, range queries are not efficient, you'll have
         3. Updates also done by appending new records so if write frequency n
            data.
      iv. Implementation details:
         1. File format: binary, so that appends are quick
         2. Updating of records: append new value at end of file instead of findi
            compaction on previous file. During this process, retain only most re
            a. length of new value may not fit old value
            b. append to file is lot faster than seek and update, in both HDD a
            c. If files are not being updated apart from append, recovery of p
         3. Deletion of records: use tombstone and during file compaction, don'
         4. Crash recovery: (assuming disks are consistent) dump RAM state to d
         5. Concurrency control: have only 1 writer thread but there can be mul
   b. SSTables and LSM-Trees (Log Structured Merge Trees)
      i. Use cases: Pretty much same as with Log Structured but little faster queries
      ii. Pros (compared to log structured)

nd Memory(M) efficiency. It's called RUM conjecture. Essentially it's saying
-conjecture/

updates/reads

to hash all keys
manages to overwhelm log compaction, then disks may fill up with duplicate

ng and replacing. There is limit on file size, when creating new file, run file
cent value for a given key. Reasons for using append:

and SSD.
partially written file is lot easier.
t bring in values with tombstone
disk so that you can recover after process crash.
tiple reader threads. Since this is limited to a single machine, it works.

s

1. Always sorted data == faster lookups
2. Can do range query
3. Uses bloom filter to avoid iops on search that is not needed
4. High write throughput

iii. Cons (compared to log structured)
1. If key space is huge, log compaction can take too long (to keep sorte
2. Since values are also stored in memory, it limits the number of keys

iv. Impl details:
1. Keep both keys and values in memory in a BST (AVL tree etc). Data is
2. When size of memtables exceeds a limit, it dumped to a file, called s
   a. Daemon thread is running compaction on segment files such t
   works best with disk IO.
   b. These files are segmented by key space range. For example ke
   us to compress these individual files and keep a sparse lookup
   to by index aaa.
   c. Merge of dump of memtables and segment files are handled b
3. For query, key is first searched in memtable then through segment fi
4. For crash recovery, (assuming disks are safe) we keep non-sorted Lo
5. This is SSTable and LSM-Tree are built on top of it, I am not sure exac

c. B-Trees

i. Usecase: used in both RDBMS and non-RDBMS. Bread and butter of DB for

ii. Pros:
1. Data is always sorted
2. Efficient reads/range queries

iii. Cons: High IOPS. Every write is done twice, once to WAL and then actual in
are called pages and size depend on underlying hardware. Since, insertions
parent will also have to be updated, further increasing IOPS

iv. Impl details
1. Only keys are in the tree and they are in sorted order. Keys are writt
logN, N being number of keys.
2. Updates: you find the key starting the search from root page and up
3. Adding key/value: you again find the right page to add this key starti
max length of keys it can support then it is split. This may mean crea
links needs to be updated too.
4. Deletion: Will generally use a tombstone to mark a key as deleted
5. Reliability: (Assuming disks are safe) most of the structure is already
execution, so in case process crashes in middle of a write operation,
recovery.
6. Concurrency: Trees data is protected by light weight locks built on t

d order)
we can store in memory

s sorted by keys. This in-memory structure is called memtable.
egment file.
hat data is sorted by keys. Use Merge sort since its sequential access pattern

ys aaa to bbb are in single file and keys bbc to ccc are in next file. This enables
map in memory. So if you need to query key aab, you know it's in file pointed

y compaction daemon. There are many details involved in this process.
iles
g Structured file for key that are still in memtable
ctly difference between them.

very long time

ndex update. Indices are stored in a fixed file size blocks (~4kB). These blocks
s are not appends and if index insertion leads to splitting of a page into 2 then

en to disk in a file size of ~4kB, called Page. Algo ensure that search is always

date the value.
ng from root page. If the page at which you have to add this key is already at
tion of new page (file). Since this page was split, it means that parent page

on disk. DB process writes the cmd to Write Ahead Log (WAL) before
it'll know what it was doing before the crash and will have a decent chance at

an of locks offered by OS

6. Concurrency: Tree's data is protected by light weight locks built on t
7. There are several optimizations, one of them is Copy-on-write. Anyti
are updated to point to this new file.

d. LSM-Tree vs B-Tree
   i. These assertions are theoretical, check empirically.
   ii. LSM-Tree:
      1. Higher sustained write throughput
      2. need to tune your db so that compaction can keep up with incoming
      3. Less fragmentation of files on disk since compaction takes care of thi
   iii. B-Tree:
      1. Higher read throughput since keys are only at one place (vs in severa
      isolation across keys.
      2. Since IOPS are too high, using SSD can be expensive since they have
      3. Files can become fragmented over time

e. Other types of indexing
   i. Secondary indices: helpful with joins can be implemented using LSM-Tree
   ii. Multi-dimensional indices: for maps or when you need to do range query a
   iii. Text search and fuzzy indices: instead of doing searching for exact text, eng
   query. Lucene uses LSM-Tree
   iv. In memory cache: RocksDb, memcacheD

3. OLAP vs OLTP
   a. Following is a quick comparison. Bottom line is you'd use OLTP for you functiona
   use case is to do analysis.

	OLTP
Read pattern	Low read QPS; Fetched by key
Write pattern	High write QPS; random access
Primary users	End users of business
What data usually represents	Latest state of events (so it's a good idea to remove h improves performance)
Data size	Less than terabytes

   b. Usually OLTP can use both LSM-Tree and B-Tree but OLAP will mostly stick to B-T
   c. OLAP
      i. Data collection ways:
         1. ETL: extract, transform, load data from disparate data sources to sing

op of locks offered by OS
me a page is updated, entire file is copied to new location and parent pages

writes. If compaction falls to far behind, reads will suffer
is

l segment files in LST-Tree). This also sets up B-Tree for transactional

more constrained write cycles than HDD but B-Trees plays well with disks

or B-Tree
cross multiple indices, see Geospatial data in Toolbox
gines like Lucene can search for indices which are with in X edit distance of a

l use cases and OLAP for analysis. The exception could be that your functional

	OLAP
	High read QPS; Mostly range query
	Usually low write QPS; Mostly bulk import: ETL/event stream
	Internal analysts
istorical data as it	Historical events
	Up to peta bytes

Tree since OLAP is query heavy.

gle DB system. This is old school technique and runs as batch job

2. Event Stream: As data is being written to OLTP systems, it generates
ii. Data warehousing: Fancy name for a DB system dedicated for OLAP. This m
iii. Schema pattern:
1. Star schema: Primary idea is that you have a table called *Facts* table
metadata tables, called *dimensions*
2. Snowflake schema: same as Star schema but *dimensions* table may f
than Star schema but Star schema is easier to work with.
iv. Storage optimization:
1. Consider following patterns in OLAP data:
a. Number of columns in *Facts* table can be several 100s and mos
b. The *Facts* table in Star/snowflake schema can have trillions of
example in Amazon.com each row may tell about order placed
2. Column-oriented DB: In regular DBs each documents stores a row wi
rows of *Facts* table you are loading all columns (100s per row) from
of storing 1 row per document, store each column in individual docu
a given rows. The assumption made here is that you don't frequently
just an append.
3. Column compression: now that we are using Column-oriented DB an
gives an opportunity to compress data in a way that don't have to de
4. Data cubes and aggregated view: I am not writing it here because it l
stopping here.

Chapter 7 Transactions

1. Failure scenarios (before, during or after operation):
a. DB software/hardware failure
b. Application crash
c. Network interruptions
d. Concurrent writes/reads
2. Transaction:
a. either a *group* of operation (read/write) on *multiple objects* will be applied entir
safety properties needed by application are still being met.
b. Sometimes transactions become the casualty of partitioning and scalability
c. Transaction guarantees: ACID however definition of ACID is database dependent
i. Atomicity
1. In concurrency: another thread will not see incomplete state of atom
isolation covers this
2. In DB: application can group multiple operations in single transaction

stream of events which update OLAP systems. Much like DDB Streams.
nay come with a UI to design ETL jobs / schemas etc.

which stores individual transactions. Columns in this table refer to bunch of

urther refer to other metadata table. This type of schema is more normalized

st of the queries will use only a subset of columns
rows but most of the columns will have a smaller number of uniq items. For
l by customer but there are only so many items and so many customers.
th all of its columns. So when you are doing a range query over trillions of
disk to RAM, although you'll need only a subset of those columns. So instead
ment. This way you can load only the subset of column you want to query for
y re-order rows and why should you? Each ETL job or event stream should be

d we know that there are only so many uniq entries in each column, this
ecompress it before reading it again. I won't go into details of this.
ooks like we are too much into weeds for OLAP already and I am fine with

ely or will not be applied at all. You can loosen the transactional guarantees if

nic operation. It'll either see state before or after the atomic operation. In DB,

n. Either all of these will pass or none. If your operations are idempotent then

you can probably give up on this guarantee.

ii. Consistency
    1. Overloaded word: consistent hashing, eventual consistency in replica
    2. In DB: essentially means that invariants are maintained in data. Thes of itself can't provide consistency. Consistency is property of applicat not contain integers etc but it's application's job to enforce consister

iii. Isolation
    1. Concurrently running transactions appear as if they were executed s Sometimes they offer weaker isolations.
    2. DDB offers *serializability* isolation when transaction is done across m items can't exceed 25 and sum of size of all items should not exceed

iv. Durability
    1. As name suggests, it's a promise that once data is written it'll not be it across DCs but data can be lost, question is how badly do you *not* v

d. Single object updates
    i. Need: failure in single object updates can leave data in weird state. These f request to update same object
    ii. These are technically not transactions, since transactions refer to *group* of
    iii. At minimum we expect serializability isolation and atomicity, usually provi Atomicity even in case when process crashes during update, DB use Write-
    iv. Usually work well with non-relational DBs where individual items do not re

e. Multi object updates
    i. Need: Same problem statement as with *single object updates* above but fo
    ii. Usually needed in relational DB or graph DB or DB which supports seconda
    iii. Fundamentally there is nothing that stops us from having strong transactio and may cost availability/performance.
    iv. Most DBs try to provide A,I,D to a large extent but if it's a distributed DB th

3. Isolation level (focusing on single node systems, see chapter 9 for multi nodes)
    a. read committed, snapshot isolation, serializability
    b. Read committed
        i. 2 guarantees:
            1. Reads will only see data which has been committed (no dirty reads).
                a. That means you'll **not** be able to see writes of a transaction tha
                b. To implement this in a way that reads are allowed while concu separately while another transaction is updating the data. Onc
            2. Writes will only overwrite data which has been committed (no dirty
                a. This is important when multiple transactions are doing multi o and y, such that transaction_1's results would be x1, y1 and tra

ation, CAP etc.
e invariants are enforced by application by means of transactions. DB in and
tion and not DB. DB can help in partial checks like column meant of date does
ncy.

serially, which is also called *serializability*. All DBs do not guarantee this level.

ultiple tables but all tables should be in same region, total number of target
4MB

forgotten. Of course, that's not true. We can persist data on a disk, replicate
want to lose the data.

failures can happen due to: process crash/network failure/concurrent

operations applied across *multiple objects* as a single unit of execution
ded by using compare-and-set operation on a single node. To guarantee
-Ahead-Log.
efer other items in same/other table.

r multiple objects.
ry indices
on guarantees in DB with partitions except for that fact that it is hard to do

hen providing these guarantees become harder.

at is going on concurrently
rrently other transactions can take a write lock, DB stores old value
e write is over, DB starts using new value.
writes).
bject updates. For example, if 2 different transaction are updating objects x
ansaction_2's result would be x2, y2. If dirty writes were allowed we may end

up in situation with x2,y1 or x1,y2 whereas desired states are e
b. This is usually implemented by taking a lock on object(s) that n
same data, it needs to wait for current transaction to either fin
ii. Need: if writes fail or partially succeed or are in progress then the data rea
from your DB
iii. Pros: simple and most basic guarantee. Provided by many DBs out of the b
iv. Cons:
1. long running transactions may see inconsistent state as concurrent w
2. No support for consistent updates in concurrent multi objects transa
c. Snapshot isolation
i. Guarantees: Each transaction reads from consistent snapshot of DB, ie, eac
even if concurrent transactions are updating this data. Essentially when a t
1. This is implemented by giving each transaction a transaction id, *txnId*
alongside. So when a *txnId=3* is reading an object, it reads data that w
value. This is called *Multi Version Concurrency Control (MVCC)*. Some
new copy is created all the way upto the root. Pages that are not upd
2. Compaction process consolidates the data when long running transa
ii. Need: In long running queries, for example, when you started the read ope
time you read object z, other concurrent transaction has updated state to
{x1,y1,z1}. This is *read skew.* What you want to read in 1st attempt is {x,y,z}
taking backups or analytics query. Also see how this affects Linearizability
iii. Pros:
1. Improvement over Read committed for long running transactions.
2. Readers don't block writers and vice versa
iv. Cons: No support for consistent updates in concurrent multi objects transa
another transaction changed that state so your inference is now incorrect.
way. *Write updates* below.
d. Serializability
i. Guarantees: Even if transactions are executed in parallel, end result will loo
1. Actual serial writes. Just do writes serially. Reads can go to other rep
you can get high write throughput. This should work well for OLTP ca
partition transactions are expensive. If your writes are complex quer
Java/Kotlin and DBs provide metrics for these. See Datomic DB. But i
different syntax and will need extra effort for deployment safety and
2. 2 Phase Locking (**not** 2 Phase Commit) or pessimistic locking, good su
a. If DB detects a deadlock, it aborts one of the transactions
b. Reads **cannot** happen while writers have lock and vice versa
c. Performance at higher percentiles may suffer
3. Serializable snapshot isolation or optimistic locking

either x1,y1 or x2,y2.
needs to be updated in a transaction. If other transaction wants to update
nish successfully or abort.
d will be inconsistent. This is most basic kind of guarantee you want to have

ox.

vrites updates the values
ctions unless application tailors the query to setup locks in right way

ch transaction sees data that was committed before start of this transaction,
ransaction starts it reads from a frozen snapshot in time.
d, which monotonically increases. Each write to object also writes the *txnId*
was updated by *txnId<3* even if newer transactions have updated/deleted the
e DBs do Copy-on-write instead, so every time a page is updated in B-tree, a
dated don't need to be copied.
ction is over.
eration and state of DB objects was {x,y,z}. You have read x and y but by the
{x1,y1,z1}, so you'll end up reading {x,y,z1}. On retry of this read, you'll see
} and in 2nd attempt {x1,y1,z1}. These errors affect long running jobs like

ctions, ie, if you read some data, make a decision and update, meanwhile
This can be avoided if application tailors the query to setup locks in right

ok like they occurred serially. Ways to implement:
licas providing Snapshot isolation. Furthermore, partition your key space so
ases since you will most likely not need cross partition transactions. Cross
ies, used stored procedures on DB, these days, you can write them in
n general stored procedures don’t have metrics and each DB has slightly
d rollback.
ummary at: https://en.wikipedia.org/wiki/Two-phase_locking

3. Serializable snapshot isolation or optimistic locking
    a. Transactions occur as if system was in *snapshot isolation* but b
    has not been updated. If it is, then restart or abort transaction
    b. If you have a system where multiple transactions frequently ov
    the other hand if that is not the case your performance will im
    c. This allows reads and writes to happen concurrently
    ii. Need: This provides utmost level of data consistency guarantees, although
    iii. Pros: Data consistency guarantees are now provided by DB and application
    application developer to craft a query so that DB can help with consistency
    iv. Cons: Performance may take hit
4. Write updates: With updates spanning multiple objects, across multiple replicas. There
options:
    a. Conditional update / compare-and-set
    b. Lock all the rows you want to update and then update
    c. Last write wins
    d. Use serializable isolation level

Chapter 5 Replication

1. Pros:
    a. Keep data close to users
    b. Handle node failures
    c. Scale out in face of high load
2. Cons
    a. Async vs sync replication
    b. Handling node failures
3. Types:
    a. Single leader
    b. Multi leader
    c. Leaderless
4. Single leader system
    a. All writes go to a leader but reads can go to replica.
    b. Sync replication: If there are network delays or even one node is slow entire syst
    have (more replicas, more time to sync) and location of replicas (further apart m
    c. Async replication: Higher write throughput since leader is not waiting for replicas
    data. Used when replicas are spread over the globe.
    d. Semi-sync replication: Only a subset of replicas are in doing synchronous replicat
    e. Node failure:
        i. Follower failure: adding a new follower is relatively simple. Bring up a new

efore committing it verifies that whatever data was read during this process
.
verwrite each other's keys then you'll have worse performance than 2PL. On
prove significantly.

performance may suffer.
n developer won't have to deal with data conflicts. Although it's still upto
y.

e are several corner cases but I won't write them here. You have following

em will have performance issue. This also limits how many replicas you can
eans more time to sync)
s to catch up but if we lose the leader, we may have lost recently committed

tion. If we lose the leader, at least we have a copy of all of committed data

machine/container get a snapshot of leader (see snapshot isolation) and

ii. Follower failure: adding a new follower is relatively simple. Bring up a new
then apply all the updates that have occurred since the snapshot was take
calls it binlog coordinates (possibly it stems from WAL of Btree)
ii. Leader node:
1. Determining leader has failed: see SWIM/phi-accural detector in Dist
2. keeping committed data safe: this is where sync/semi-sync replicatio
3. electing new leader: candidate must be the one who has most recen
4. broadcasting new leader to the system: update DNS when election is
5. Avoid old leader from incorrectly assuming that its leader: Raft solve
6. Worst case, Put human in the loop 😕
f. For replication ship data directly from leader to node eg. WAL from BTree or LSM
engine
5. Managing replication lags
a. Reading your writes: this is important esp. When user updates data and refreshe
others to see it bit later. How do we ensure this?
i. If this is reader's profile page, then query from leader. To scale this up use
ii. If client app knows when the last update was it can send that as a param in
information is not available. There is no guarantee that sibling has the info
iii. Replica can tell client when the last update to data was made and app can
iv. If replicas as spread across globe use write through edge cache.
b. Lags across replicas for different partitions: it's theoretically possible to have cau
partitions increase and number of replicas in each partition increase. So most sys
DDB.
6. Fixing Conflicts
i. [data loss] Last Write Wins (LWW), some way for figuring what is "last"
ii. Keep the conflicting data and ask user to resolve it or give user a chance to pick w
iii. Keep operation on data commutative, associate and idempotent so eventually st
iv. custom logic that is uploaded to db as script to merge conflict
v. Version vector:
i. Each replica keeps track of version of key and also track of version of key fo
ii. When client reads, each replica send back data with version number and cl
iii. During write, client needs to send the last version number read so replicas
7. Multi leader replication
a. Difference between algo like Paxos and Multi leader system is that in multi leade
all/subset nodes come to a consensus
b. Benefits and how single leader system can do the same thing:
i. Most impactful use is low latency geographically distributed writes. In sing
have shards located globally. All shards don’t have to be in single DC but cr
ii. Scalability: true but in single leader you can use shards to scale indefinitely

machine, container get a snapshot of leader (see snapshot isolation) and
n. DBs like MySQL allow you to get updates after a certain udpate. MySQL

tributed system
on comes into play
t data (Raft does it by log entry index + session id)
s over/use zookeeper/external kv pair system
s this by sessionId and log entry index

MTree, although this may mean that replication is dependent on storage

es page, they should see the updated data, although it's probably fine for

shard the users
n query, forcing replica to route request to another sibling if latest
rmation either and eventually this request will probably end up with leader.
tell user to refresh to see more recent updates.

sally consistent read cross partitions but it becomes expensive as number of
stems don't support it (eg. Vitess) or partially support it, eg transactions in

which write to keep. Maintain all history.
tate converges (CRDT/operational transform)

or other replicas
lient needs to do the merge
can figure concurrent writes

er each leader is free to make any commitment to the client whereas in Paxos

le leader, You can choose to have write thru edge caches like Slack does or
ross shard queries will screw up, even more than they were before.

iii. high availability: Master failures can be done quickly using systems like zoo
iv. Tolerance of network problems: fair enough but you really want this so ba
v. Vitess does not support multi master instead offers this

c. Problem:
i. conflict resolution. If your operations are commutative, associate and idem
ii. DDB global table handles this by LWW

d. Use cases (these use cases are forced and there is no way out):
i. Offline updates: calendar app working offline on user's device acts a maste needs to be resolved manually
ii. Collaborative editing: if users are updating offline then it falls under multi l perfect. Manual conflict resolution with feature of rolling back changes.

e. Topology of leaders is yet another variable and I wont talk about it because all th

8. Leaderless replication
a. None of the nodes decide what will be committed. Application logic is free to cho
i. Advantage: client can decide how strict quorum they need/want. This dicta
ii. Disadvantage: clients have to figure out conflict resolution.

b. Quorum:
i. if n is total nodes (odd number) and w is min number of nodes to ack write
ii. Sloppy quorum and hinted hand-off
1. Incase of network partition, replica nodes may not be available. Read available nodes to replicate data. This is sloppy quorum.
2. When network heals, this replicated data is put back in nodes where hinted hand off and sloppy quorums, see DDIA or DHT

c. Writes and replica catchup
i. Application logic tries to get a quorum on nodes
ii. If subset of nodes miss the write they are caught by one of the following
1. Anti entropy: daemon that brings all replicas to sync writing latest ve
2. Read repair: when application reads and discovers that a subset of re

d. Edge cases with quorum
i. Concurrent read and writes may have edge case where some subset return nodes are stale
ii. If write to quorum failed and then rollback of these partially committed no
iii. DB itself does not provide any isolation guarantees, it's upto application to

e. Example: Amazon Dynamo

Chapter 6 Partitioning

1. Pros: add scalability to system; often used with replication to improve durability

okeeper/raft and for reads, replicas are always available
d?

npotent then you don't have to worry about this.

er and another master is in DC. Upon syncing, there will be conflicts which

leader use case. CRDTs/operational transformation may help but they are not

his multi leader talk is just not practical.

oose how strict/loose it wants the read/write quorum.
ates how good the consistency will be.

e and r is min number of nodes to respond to read then w+r>n

ds can still go thru but for writes, the co-ordinator node will pick other

it belongs, this is hinted handoff and is used in Dynamo: Dynamo employs

ersion of an object to all replicas
eplica has old data, application pushes an update to old replicas

ns latest and rest returns old value and reader maybe lead to believe that

odes failed, system is in inconsistent state
figure this out.

2. Cons: cross partition queries are either poorly supported or not supported
3. Cross partition transactions
    a. DDB: limits reads/writes to 25 items and there is limit on size of query and all tab
    b. Vitess: cross shard reads may not be consistent. You can force it to do 2PC but th
4. Partitioning key-value stores
    a. Partition by key range. Each partition owns a range of *keys*.
        i. Pros: range queries are quick since data is always sorted (LSMTree does thi
        ii. Cons: there can be access pattern where you end up with hot partitions. Al
    b. Partition by key hash. Each partition owns range of *output of hash function*
        i. Pros: easy to relive hot partitions
        ii. Cons: to do range query you'll have to search all partitions
    c. Partition by compound keys {hash key, sort key}
        i. Pros: can do range query for given hash key without leaving the partition
        ii. Cons: range query on hash key is still expensive
5. Handling hot spots: If a given subset of keys are being accessed frequently here is what
    a. Put those subset of keys in a separate partition so they get dedicated machine po
    b. Append they key with 2 digit random number. This splits the key into 100 keys w
    query multiple partitions and keep track of which keys have been split.
6. Secondary indices:
    a. Usually wont identify a record uniquely.
    b. Don't map to partitions cleanly (of course)
    c. Implementation
        i. Local index:
            1. Each partition keep tracks of objects that belong to certain value of s
            2. During query, DB does scatter gather call across all partitions. This is
            3. Updates to index are relatively quick since the partition which owns
        ii. Global index:
            1. The index itself is partitioned. So a particular key value will appear on
            2. Actual document and global index key value may live of separate par
            3. Reads are not subject to tail latency amplification
7. Rebalancing partition
    a. Need:
        i. Relieve hot partitions (also see Handling hot spots, above, on how to split a
        ii. Replace broken nodes
    b. Minimum expectations:
        i. Read/write should keep going uninterrupted
        ii. Load per node is now either better or at least same as before
        iii. Minimal movement of data across nodes and/or partitions
    c. [not recommended] hash mod N

bles must be in same region
hat drops performance by 50%

is)
lthough there can be automations to address this.

t you can do:
ower
which will now to go to different partitions. Down side is that you'll have to

secondary index.
prone to tail latency amplification.
the document that is being updated also has the copy of index.

nly in single partition.
rtition so updates may take some time to propagate to global index

a single key across partition)

c. [not recommended] hash mod N
    i. N is number of nodes and you are dividing entire hash range equally acros
    ii. Problem is an N changes (and it will due to scaling) the range that each no
    partition that is not a good idea.
    iii. DHT has similar problem, as explore in Dynamo paper: Fundamentally, dat
    joins/removals: Dynamo employs hinted hand off and sloppy quorums, see
d. Fixed number of partitions
    i. Useful when you know size of data will not change too much over lifecycle
    ii. Start with partition count of 10x the number of nodes
    iii. Pros: you'll now need to move partitions across nodes and not keys across
    iv. Cons: if data set size varies hugely during application lifetime, having fixed
    load may be distributed unevenly, how ever if you have a hash function wh
e. Dynamic partitioning
    i. Useful when size of data may vary a lot over lifecycle of application
    ii. Pros: As data grows, DB decides how partitions will be split and account fo
    iii. Cons: Complex to implement
f. Partition proportional to node count
    i. Each node gets fixed number of partitions. When new node comes up, som
    half and old node owns other half
    ii. In long run, random split of partitions and up distributing work fairly
8. Request routing
    a. Use case: when a client wants to read/write a key, which node should this reque
    b. Ways of routing request
        i. A fleet of request routers which "knows" which partition key belong to an
        ii. Every node in fleet "knows" about every other node. Now, Client goes to a
        iii. Clients are "aware" of partitioning and know which node to go to
    c. The "know"/"aware" part is critical and unless all parties are not on same page, a
        i. Use Zookeeper to track partition assignments and keep hash function avail
        ii. Use gossip among the DB fleet to sync everyone with latest state of the wo
        iii. Use consensus algorithm (paxos) for deciding primary and secondary owne
        to get new one.

Chapter 9 Consistency and Consensus

1. Consistency guarantee types:
    a. Eventual consistency: replicated DB will converge if no operation are performed

s nodes
de owns will shift and that will mean that Keys will have to move across

a partitioning and data placement is intertwined, i.e and handling node
e DDIA

of application

partitions; Splitting/merging partitions is hard, we are avoiding that
number of partitions may be painful; If you got partition boundaries wrong,
hich can take care of this, you should be fine

r evenly distributed work load across nodes.

ne random set of partition is chosen and split into half. New node owns one

st go to?

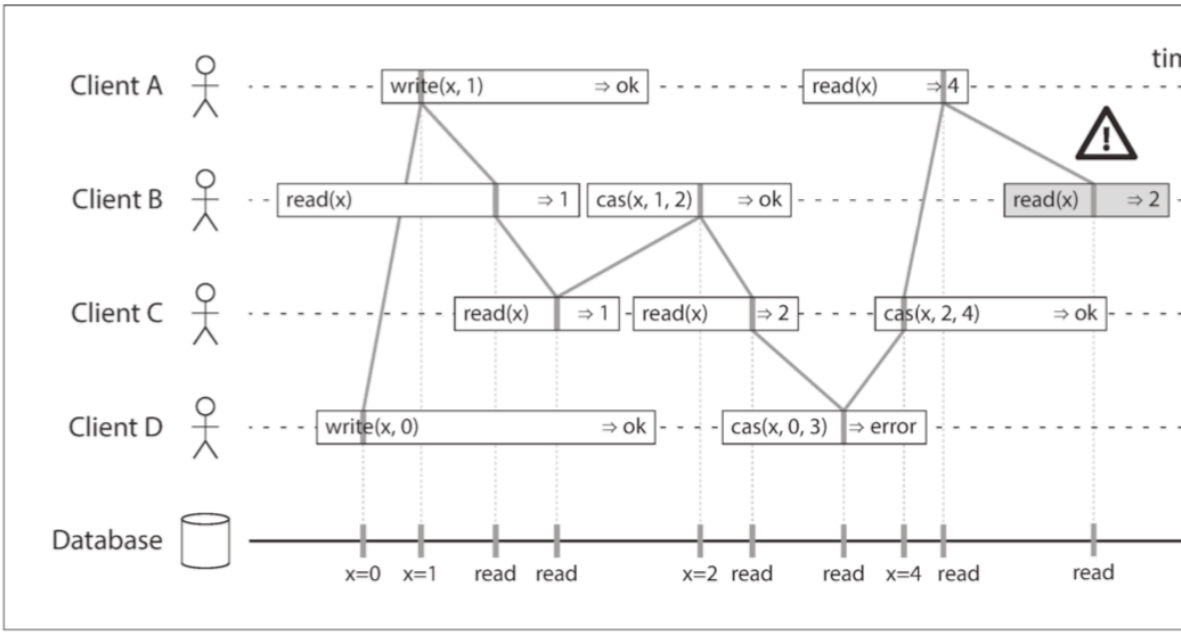
d which node partition is on
random node in fleet, and this node redirects request to correct node

availability will take hit. Following are ways to accomplish this:
lable to all. Kafka uses this to keep track of partition owners.
orld
ers of a partition. Use SWIM to detect that previous owner is dead and need

in some unspecified time. This is weak guarantee

a. Eventual consistency: replicated DB will converge if no operation are performed
b. Causal consistency (Partial order): as provided by Snapshot isolation. Partial orde updates to different keys. That means there can be 2 operations where system c
c. Linearizability (Total order): discussed below. Total order because for any pair of updates to different keys are forced to go one after other.
d. Stronger consistency comes at cost of performance. The more distributed nodes you'll have to go to weaker consistency to have reasonable performance since ne Most systems should work well with causal consistency. People are still working systems. DDB supports both eventual consistent and causally consistent read: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowIt

2. Linearizability
   a. Making replicated DB look like it has single node. So there is no lag in consistency sending updated value to a concurrent read req) all future req. will see this value may look like following. Look how final read done by B does not follow *Linearizab*



   b. To prove that black box is Linearizable you can record timing of all req/responses though.
   c. Comparison with Serializability:
      i. Serializability means that it'll appear as if concurrent transactions were exe concept of grouping operations together as transaction.
      ii. 2PL and Actual serial execution are fully compatible with Linearizability. Be past transactions are done. This holds true of multi replica DB which offers
      iii. Serializability Snapshot isolation (SSI) however is incompatible with Linear for read only transactions. Consider a long running backup job. While this t concurrent transactions which would have updated the data. When this jo
   d. Use cases
      i. Zookeeper has Linearizable writes but causally consistent reads: https://zo

in some unspecified time. This is weak guarantee.
er because system allows concurrent updates with read or concurrent
annot tell which happened before.
operation there is a strict happened-before relationship. It appears that even

(all nodes in same AZ, all nodes in same region, nodes spread across globe)
etwork delay will increase in proportion with distance on public internet.
on getting causal consistency with performance of eventually consistent

Works.ReadConsistency.html

y ie, once DB says it has written something (either by ack'ing write req or by
e. This is a recency guarantee. When Concurrent updates are made system
bility since prev. Read by A gives value 4 and there were no further updates:

s and see if they can be arranged in sequential order. Syncing time is a issue

ecuted serially whereas Linearizability is recency guarantee, there is no

ecause in both of these systems, future transactions are forced to halt until
these Serializability guarantees.
zability. SSI allows concurrent read/write and behaves like snapshot isolation
transaction is running, it is reading a snapshot from past while there may be
b finishes, it'll see old (although consistent) data. This violates Linearizability

ookeeper.apache.org/doc/r3.6.2/zookeeperInternals.html#sc_consistency

i. ZooKeeper has Linearizable writes but causally consistent reads: https://zo
ii. Leader election in distributed system, else you'll end up with split brain.
iii. Distributed locking, I mean if multiple nodes are trying to acquire lock and
iv. You'll note that cases where 2Phase locking or actual serial execution is de
v. Need for Linearizability is most noticeable when an actor triggers message
an image on HDFS and in parallel send a msg via Kafka to a compression se
you'll have inconsistent results. There are many ways to work around this.

e. Implementation
i. Single leader and multiple replicas (potentially Linearizable): If all replicas s
new leaders are elected, then this DB is Linearizable
ii. Consensus algorithm (Linearizable): If we strictly go for Linearizability then
iii. Multi leader replication (NOT Linearizable): Leaders by definition have pow
multileader. Given that there are multiple leaders, conflicts will be commo
iv. Leaderless replication (probably NOT Linearizable): these system work on q
which throws Linearizability out of window.
v. "Compare and set" or Last Write Wins operations without consensus/2Pha

f. Cost of Linearizability
i. In systems communication over networks, choosing Linearizability means p
Partition is almost always a given.
ii. In multi CPU but single node systems, each CPU makes multiple copies of d
in CPU_2 for a while or use language semantics to sync the data like *volatil*
Linearizability but it improves performance.
iii. Linearizability kills performance. There are often ways you can use weaker

3. Ordering
a. Need:
i. In context of snapshot isolation
ii. Detecting concurrent operations
iii. Consistent prefix read: this is weaker guarantee than *Read Committed*. Thi
row (causally consistent)

b. Causal order vs total order:
i. In real world the limit on speed of light keeps things causally (not Total) co
which may screw up causality.
ii. Total order (Linearizability): any 2 operation can be arranged in *happens b*
iii. Causal order: there can be 2 operation where there is no strict *happens be*
between operations.
iv. In most cases you don't need Total order/Linearizability. Causal ordering is

c. Enforcing causality
i. Single leader system: pretty straight forward since only one node is respon
isolation has optimistic locking (see Serializable Snapshot isolation), that is

oKeeper.apache.org/doc/r3.6.2/zookeeperInternals.html#sc_consistency

system cannot guarantee Linearizability then locks are moo point
sired, Linearizability is also desired that is perhaps they go hand in hand.
s to different systems regarding same thing. For example, app server putting
rvice. If compression service tries to read the file before its written/updated,

should always be in strictly in sync and there are no bugs/split brain when

that means we chose CP system which impacts availability.
ver to make unilateral decision. If they are trying to do consensus, it's not
n, don't even think of Linearizability or even strong isolation for that matter.
quorum, that usually means that a subset of nodes will not have latest data

se lock/actual serial execution do not guarantee Linearizability.

picking Consistency over Availability. That's the cost because network

data, in L1, L2 cache and in RAM. Data in CPU_1 may not be visible to thread
le in Java or locks. Default choice having multiple copies means no

consistency guarantees for what you need to achieve.

s means that before you see an update to a row, you'll see creation of that

nsistent. In computer network there can be random delays, GC pauses etc

efore relationship
fore and these operations are concurrent. This defines a partial order

enough. This is good because Linearizability is expensive.

nsible for committing writes. You don't have to full 2PL Nazi for this Snapshot
another way of keeping track of causality

ii. Multi leader/leaderless systems
    1. Lamport timestamps:
        a. Each node maintains a version of key. This version has nodeId clock, instead it is incremented on every update.
        b. Not going into details because I don't think it's worth remembe
    2. Version vector vs Lamport timestamps: Version vectors are way to d total order
iii. Enforcing causality is not same as ensuring uniqueness of data. For exampl of events but that guarantee does not offer a way to stop users from signi after operations have been executed not during the execution. Total order

d. Total order broadcast: msg delivered exactly once to all nodes
    i. Need:
        1. In multi leader/leaderless systems just knowing total order may not
        2. DB replication, you need all replicas to apply updates in same order
        3. Serializable transactions are also a type of total order broadcast
        4. Leader election is distributed systems. See Single leader replication i
        5. Consensus services like Zookeeper have total order broadcast
    ii. Invariants that need to be enforced:
        1. Reliable delivery: if a msg is delivered to one node, it's delivered to a
        2. Totally ordered delivery: msgs are delivered to every node in same o
    iii. Comparison with Linearizability
        1. Total order broadcast allows that some nodes may lag while getting
        2. You can build write Linearizable systems on top of total order broadc some nodes may be lagging behind. See *managing replication lags* in
    iv. Total order broadcast is equivalent to repeated rounds of consensus

4. Distributed transactions and Consensus
    a. Need:
        i. Leader election in distributed systems, for eg which node will lead a partiti
        ii. Atomic commit: transactions across multiple partition/nodes which are no what will be committed. Leader may wait in sync until a subset of follower have failed.
    b. Distributed transactions
        i. 2 Phase Commit (distributed atomic commit)
            1) There are multiple nodes that needs to commit and a Transaction M
            2) Process
                a) TM generates a globally uniq transaction id for a commit and u
                b) TM sends prepare request to all nodes and awaits "yes" from a
                c) Node response (Phase 2)

and monotonically increasing counter. This counter does **not** come from

ering.
etect if updates were concurrent whereas Lamport timestamp are to enforce

e, 2 users can sign up with same userid in a DB that can guarantee total order
ng up with same userid. Essentially, you'll know the total order across system
broadcast is way to fix it.

be enough if you want to guarantee uniqueness of key/value across fleet.

s same as total order broadcast but

ll nodes
rder.

updates where as Linearizability is recency guarantee for read.
cast systems. To ensure read Linearizability, you'll have to extra work since
Replication chapter.

on of DB with multiple replicas
t in leader-follower hierarchy. In leader-follower hierarchy, leader dictates
s commit but followers don't get a chance to say no to commit, unless things

anager (TM)

ses that to track status from nodes
all (Phase 1)

i) all nodes respond with "yes" in bounded time. This is po
crashes after saying yes, TM will retry forever. Wheneve
yes.
ii) Any one node fails to respond or does not respond with
3) Pros: more consistency than just a commit command or Quorum ser
4) Cons:
a) Node may respond with yes and then crash and TM retries ind
b) This is a blocking commit if TM crashes.
i) TM may crash after it has sent commit request to subset
transactions.
ii) Nodes cannot willy nilly abort a long pending transactior
change until TM says so.

ii. 3 phase commit (distributed atomic commit)
1) Assumptions:
a) Network with bounded delay
b) nodes with bounded response time.
c) Perfect failure detection, since this does not exist 2PC is more
2) Pros: non-blocking commit with same consistency guarantee as 2PC
3) Cons: since this relies on "perfect failure detection", it's not practical

iii. Distributed transactions in practice
1) X/Open XA (eXtended Architecture) is API to support 2PC by providir
2) 2PC impl
a) If TM has lost it's WAL, there need to be manual intervention.
b) TM becomes Single Point of Failure - use replication for TM wit
done automatically by Paxos or Raft at which point you can asl
3) If part of systems fail, transaction will either fail or be stuck indefinite
4) Distributed transactions are hard but there are ways around it (chap

iv. DDB's approach
1) Source: Transactions and Scalability in Cloud Databases—Can't We H
2) Key take away (or how if I were to do this on top of memcacheD clus
a) Partitioning and replication: For each partition let there be a le
b) TransactionManagers: stateless fleet of servers that execute tr
stamp. There are 2 key things:
i) TMs decide between 2 concurrent transaction which one
ii) TMs need to agree on monotonically increasing timestar
c) Next steps are similar to 2 phase commit. TMs figure out leade
and if all are ready then commit.
d) TMs have a commit log, so in case TM drops dead a daemon m
e) If storage node falls dead well then the system which bring up

int of no return. TM writes decision to WAL and sends commit. If a node
r the node comes back up, it needs to commit this message since it has said

yes then abort
nt across nodes

efinitely

t of nodes. When TM recovers, WAL can be used to commit/abort pending

n because they do not know state of other nodes, so the state must not

widely used.

l to implement it.

ng an interface of Transaction Manager. Java has JTA which implements it.

Rebooting nodes will not help since correct impl. Will boot into same state.
th single leader and multiple follower. Leader election can be manual or be
k why not use Paxos?
ely.
ter 11 and 12)

Have Both?
ster what is needed
eader and its followers
ransaction. Each transaction gets a strictly monotonically increasing time

e should be executed first based on which transaction reached TM first
mp - possibly via consensus - presenter did not clarify how
er of each partition to which each key belongs to. Ask them for getting ready

nonitoring the commit log will do clean up.
partition healthy will kick in.

f) Note that leader of individual partition does not know what ot

c. Fault tolerant consensus
   i. Difference from distribute transaction(DT): TM is SPOF in DT, so if TM fails long as majority of nodes agree. For details, see Consensus on Transaction
   ii. Invariants in consensus (assuming no Byzantine faults)
      1) Uniform agreement: no 2 nodes decide differently
      2) Integrity: no node decides twice
      3) Validity: if a value was decided, some node would have proposed it
      4) Termination: every node that does not crash, decides some value (as requirement
   iii. Total order broadcast properties:
      1) is equivalent to repeated rounds of consensus
      2) All nodes decide to deliver same msgs in same order (Uniform agree
      3) Msgs are not duplicated (Integrity)
      4) Msgs are neither corrupted nor fabricated (Validity)
      5) Msgs are not lost (Termination)
   iv. Single leader replication is same as total order broadcast but for total orde
      1) In algo like Paxos and Raft, each time a leader is elected a new term monotonically increasing integers.
      2) If old leader shows up and tries to do a commit, there will be term m
      3) If existing leader does fails to send heart beat to more than half of fo becomes new leader. This may look like 2PC but biggest difference is 100% of nodes in 2PC.
   v. Detecting node failure based on time out can be tricky esp.
      1) Same DC or worldwide
         a) Nodes spread over the world: tweak timeout for heartbeat as thresholds adapt as time passes.
         b) Nodes in same region/DC: SWIM/gossip protocol: quick overvi
      2) Same DC/region only
         a) In GFS/HDFS nodes send heartbeat to name server
         b) Using zookeeper/consul/etcd but zookeeper itself may fail:
            i) different sets of nodes are in different partition and zool
            ii) Zookeeper is in isolated partition
            iii) Zookeeper can talk to only a subset of nodes but other n
         c) Getting simple metrics from node (Prometheus pulling or a sid observers of that node can see.

her partitions are doing. It just knows about the key it owns.

progress is halted. On the other hand in consensus (Paxos) do not block as
Commit by Lamport, Gray

ssuming that more than half of nodes are available). 2PC does not meet this

ment)

r broad cast you need single leader. There is circular dep.
starts and all new commits will start from 0 under this new term. Terms are

ismatch and it should fall back as follower
llowers, then these followers start a election and whoever gets most vote
s that here you need support of just majority to make progress as opposed to

time progresses, Phi Accural detector. The decision is not binary and

ew

keeper can see both sets

nodes can communicate
e car pushing to central system) may be too granular and may not show what

Chapter 10 Batch Processing

1. Types of request/response systems:
    a. Online services: you send a req and expect response in few seconds
    b. Batch processing: you send a req and system may take several hours/days deper
    c. Stream processing: sits between online systems and batch processing
2. Key property: In batch processing, input data is bounded
3. If you are sorting large amount of data in memory, you can use same pattern as LSM-T compaction job will run merge sort. SSTables and LSM-Trees (Log Structured Merge Tre
4. Map reduce:
    a. Also see notes from paper: MapReduce (Jeferry Dean et. al.)
    b. Data is read from and written to distributed filesystem (GFS/S3/HDFS)
    c. Difference between HDFS and NAS(Network attached storage) is that HDFS is bas however NAS is shared-disk system where central storage appliance is used by m hardware, ie vertically scaling.
    d. Sort-merge join
        i. Need: if you have a log file with IDs and you need some data correspondin network call to db/remote system to get associated with IDs
        ii. You create a secondary map job that takes dump from remote system and You can do it in a way such that the data process by this job and primary jo and does not need to do calls to remote system every time. This is called so together sorted lists of records from both sides of the join.
    e. Handling hot key
        i. Need: there may be systems where a given key is hot that single reducer w
        ii. Either pre-analyze the data (Pig workflow engine does this) for hot key or a
        iii. When a mapper sees hot key in its partition, it sends it to a random mappe Reducers will do what they do and then this output can be sent to another
    f. Map-side join vs reduce-side join
        i. Reduce-side join: when reducer does the *joining*, like in sort-merge join, it'
        ii. Map-side join: in this case Mapper does the join, following are usecase/typ
            1) Broadcast hash join: if the remote data (like the DB query we did in s do the join. Word *Broadcast* essentially means that each mapper has
            2) Partitioned hash join: same as Broadcast hash join but instead of sto needs for the partition it is working on
        iii. When to use which:
            1) Map-side: makes more assumptions about size/sort order of data. Th

ding on size of data

ree. Keep active set in memory and push rest of it to files and background
ees)

sed on share-nothing principle (ie, horizontally scalable and no SPOF)
ultiple systems which although may be using RAID and some custom

g (say, name) to these ids in end result, your mapper/reducer will have to do

puts ID -> name map in same partition as your primary mapper would have.
b are consecutive. Now when you Reducer runs, it knows the juxtaposition
ort-merge join since mapper output is sorted by key and reducer then merge

ill take too long to process
sk user to give hot key as input.
r for processing. Essentially distributing work for hot key across nodes.
mapReduce job which collects all the data related to hot key together

s called reduce side join
es
ort-merge join) can be loaded in memory of each mapper. Mapper itself can
this data and *hash* refers to in-memory hash table to store this data
ring entire join data in memory, mapper knows how to get just the data it

e output of mapReduce is partitioned and sorted in same way as large

output
2) Reduce-side: makes less assumption since mapper puts all the requi by join key

g. Use cases of mapReduce
   i. Creating readOnly search index:
      1) there are DBs (like Vodemort) which support creation of index from
      2) It is also possible to create incremental search index, Lucene does th
   ii. Setting up data for Massively Prallel Processing (MPP) systems or Data Wa MapReduce comes in picture so that schema updates do not come in way the data in DataLakes (S3/HDFS) then a map reduce job does the ETL part. issues.
   iii. MapReduce yields itself to feature engineering in ML, not sure exactly how
   iv. OLTP system like Hbase and OLAP system like Impala build on top of HDFS.

h. Other properties
   i. mapReduce jobs are clean all-or-nothing systems. Each individual mapper/ fails.
   ii. It's human fault tolerant: if code has bug, rollback and re-run the job as op bug (in most cases)
   iii. Same set of input files can be used to infer different things. So you don't ha HDFS/Hadoop have the property of keeping compute close to source of da
   iv. MapReduce by design can tolerate high failures of independent tasks

5. Dataflow engines and MapReduce:
   a. Need: Often times multiple mapReduce tasks are chained one after another. This which will be consumed by next job. Issues are:
      i. Too many intermediate files are stored on disk
      ii. Mapper of next task could have been merged with reducer of previous tas
      iii. Mapper of next task cannot start until previous task's reducer is done (ie n
   b. Dataflow engine: They model entire job as single workflow as opposed to indepe because:
      i. If MapReduce was a 2 step function (map followed by reduce), then workf structure these functions in a Directed Acyclic Graph (obviously)
      ii. Function which is next in chain does not need to wait for all of the data to (synonymous to streaming)
      iii. Since engine already knows which functions have to be applied to what da than regular MapReduce and less intermediate files are created. Since no i failure. There are 2 options
         1) Although entire pipeline can be run from beginning but that's waste depends on time of day/the order in which data is read from hashma data needs to be restarted. It's lot more easier if we maintain detern

red data for Reducer to use. Output of mapReduce is partitioned and sorted

output of mapReduce. While new index is being formed, DB serves old index.
at.
rehouse(DW): OLAP systems like MPP or DW often have a fixed schema.
of transferring data from OLTP system to OLAP system. OLTP systems dumps
This reduces backpressure on OLTP system in case system doing ETL is having

.
Neither of then use MapReduce.

/reducer may have run multiple times but you either get full output or job

posed to in regular DB rolling back code will not undo the damage done by

ave to move data around, just update the code logic and get new results.
ta, ie, literally the same node. So no network calls to fetch data.

s chain can be 50-100 in recommendation systems. Each job is creating file

k
o streaming)
endent subjobs. Spark/Flink are well known for this. They are lot faster

low engines have generalized it to N-step function and workflow engine

be processed from previous step, instead it can work on whatever is available

ta, they can plan and optimize execution, as a result, execution is lot faster
ntermediate data is written to disk, we risk fault tolerance in case of node

of time. Another problem is that if output is non-deterministic, say it
ap etc then restart would mean that any other function working on same
inism or idempotence

data needs to be restarted. It's lot more easier if we maintain determ
2) Flink does checkpointing of data and other engines have similar bala
6. Graph DB and batch processing
   a. Pregel: MapReduce does not make much sense incase of GraphDb. Instead mode sends "messages" to next connected vertex. Each vertex maintains its state. Whe processed input to next vertex. If there is no input, there is no processing needed Vertex as in vertex of graph.
   b. Pregel guarantees:
      i. all messages sent in one iteration are delivered in next iteration. So all prio
      ii. Each message will be execute only once by the target recipient vertex. This distributed file system.
      iii. Idempotence is again a good property to have to recover from faults.
   c. Pregel is distributed but since there is no simple way of partitioning graph vertice one vertex to next. That can become a bottleneck.
7. Marriage of Declarative programming and MapReduce: Dataflow engines provide the A execute the query in efficient way. This almost looks like declarative programming whi efficiently while still having the ability to run arbitrary code (like MapReduce) on arbitr

Chapter 11 Stream Processing

1. Transmitting of events
   a. Events are to streams what record are to batch processing systems - small, self c happened in some point in time.
   b. Related events are often grouped together as topics/stream
   c. Messaging system are built as publish/subscribe systems, 2 biggest questions are
      i. What happens when consumer cannot keep up with incoming messages. C
         1) Drop the message (like in case of sensor metrics)
         2) Consumer applies back pressure (via TCP etc) and producer tries to h application.
      ii. What happens if consumer is unreachable. From producers perspective it's pressure is measured (TCP timeout vs response of RPC msg)
   d. Message passing w/o brokers
      i. Usually UDP is used and few dropped packets wont make much difference in case client expects re-transmission.
      ii. ZeroMQ is brokerless and does this over TCP
      iii. Sometimes consumer may expose a port to which producer can directly pu
      iv. Although these work but problem here is fault tolerance and fanning out m consumers are always online.
   e. Message passing w/ brokers

nimism or idempotence.
nces built in.

el followed here is much more similar to Actor model (like Akka). Each vertex
en executor executes a given function on the vertex this vertex forwards
d. This is called Pregel model. Note that by vertex I do **not** mean a function.

r iterations should finish before starting next iteration.
s is done by checkpointing state of all vertices at end of each iteration on a

es there can be too much of network communication to pass messages from

API to write relational style building blocks and engine figures out how to
ch makes it easy for users to program the system and system executes tasks
ary data format (data lakes)

ontained, immutable object that contains details of something that

e:
Option are:

hold msg. What happens when producer reaches its hold limit depends on

s similar to consumer not being able to keep up, depending on how back

and there may be an expectation that producer knows some history of data,

ush msg to via RPC
multiple consumers. There is implicit assumption that producers and

e. Message passing w/ brokers
    i. These are specialized DBs to which multiple producers and consumers can
    ii. Fault tolerance is now handled by these systems. Producer usually just loo
    has a configuration to keep msgs in buffer for so long.
f. Message broker (MB) vs DB
    i. Message deletion: MB delete msg as soon as consumer gets it, DB wait for
    ii. Data size: MB works on relatively smaller data than DB, keywork is *relative*
    iii. Querying data: MB supports subscription based on some message pattern
    iv. Result of query: MB does not guarantee any isolation per se, they may offe
    ACID
g. Multiple consumers: there are 3 options
    i. Load balancing: msgs are balanced b/n customers. This is used when its ex
    ii. Fan out: each msg is received by all consumers
    iii. Mix: group of consumer gets all the msgs for a stream but each consumer i
h. Consumer fault tolerance
    i. If there is strict requirement of msg delivery then consumer should send ba
    removed. SQS has default hidden timeout of 30s
    ii. This means that if your queue is FIFO then broker will wait until current ms
    FIFO queue
i. Log based brokers:
    i. Log vs queue based brokers: SQS is queue based broker vs Kafka/Kinisis are
    ii. Need: if you want your consumers to have some visibility into historical ms
    can rewind back the offset (till certain limit)
    iii. Essentially each msg is appended to log and logs are partitioned for horizo
    the length of the logs.
    iv. Its hard to guarantee sequence of msg delivery across partitions.

2. DBs and streams:
    a. Need:
        i. replication in DB or state machine replication in consensus algo or total orc
        ii. Updating caches/search indices/Data warehouse etc as DB is being written
    b. Change data capture
        i. if every DB update can propagate its WAL or new events to downstream sy
        updates. There are systems like FB's wormhole or DDB's streams which ger
        data. This idea is called *Change Data Capture* and is not very well supporte
        via API. It essentially means that a DB will allow other to look at its WAL dir
        ii. While streaming DB WALs it's a good idea to have same interface as Log ba
        Kafka/Kinesis and downstream services will pick it up from there.
        iii. Log compaction: If a downstream system has to do a cold start, it doesn't h
        state of each key. This is where log compaction comes in. A background jol

push/pull data from.
k for an ack from broker and when consumer applies backpressure, broker

explicit delete cmd
ly
but DB support well formed queries based on secondary indices/joins etc.
er FIFO or atleast/atmost once delivery but DBs are on entire spectrum of

pensive to process each msg

in that group may get only a subset of msgs.

ack an ack to broker that it has finished the msg and msg can now be

sg has been ack'd for processing, that impacts throughput else don't have

e log based.
sgs, you want log based broker. Each msg has a offset in log and consumersa

ntal scaling. Each msg gets monotonically increasing offset and you configure

der broadcast. They are all use cases for streaming writes across nodes.
to

ystems like caches etc. then it'll be lot more easier to keep up with data
nerate events for each update and downstream system can use it to update
ed natively in older systems. However newer systems do expose their events
rectly without having to use SQL etc.
ased broker. Since WALs are logs. One of doing this could be to send events to

have to go thru every update done to every key. It just needs to know latest
p maintains latest state of DB and this is used for cold starts.

c. Event sourcing
    i. Contrary to change data capture we can capture events at application leve
    ii. Pros: its easier to do RCA or replay events. It makes it easier to reason abo
    iii. Cons: log compaction at event level is harder because by just looking at ev
    from event, you need to execute application with that event (and prior sta

3. Event processing
    a. Need: Fraud detection / trading / monitoring systems
    b. Types:
        ★ i. Mental model: there is critical change in mental model. In DB, data is persi
        are persistent and data keeps changing. So anything that requires
        ii. Complex event processing:
            1) these systems store queries and monitor streams. If a part of stream
            2) Running some analysis and updating some statistical model
        iii. Using streams in place of RPC for communication.
    c. Time: timestamp of events, there are following options
        i. Use end users time (may have clock drift)
        ii. Use time when events were sent to server. If client was working offline/se
        iii. Time at which event was sent to server according to client clock
        iv. If we subtract last 2 times, we get offset between server and client clocks.
        Assumptions are that client's clock is not drifting and there is no network d
    d. Aggregation windows: you'll usually bucket events by time windows and then ag
        i. Tumbling window: Each event belongs to exactly 1 window
        ii. Sliding window: Event may belong to several windows, as window slides in
        iii. Session window: Events that occur together are group together. For examp
        by another period of activity of 2 hours. We have 2 buckets of activity. This
        iv. Book is confusing between sliding and hopping - I am glossing over it - not
    e. Stream joins
        i. Stream-stream: you have events from 2 streams coming in. There is a uppe
        example joining a search result and actual click on search will work if both
        join.
        ii. Stream-table: as name says, you are looking up a DB table for stream even
        iii. Table-table: as a table is being updated, we use those updates to update o
        followers of this user should get this tweet in their timeline. We maintain t
        iv. As you can see there is dependence on what time does system see updates
        with event that is being processed, how do you handle that? Usually the ev
        state of DB table. Downside is that you need to maintain historical state in
    f. Fault tolerance
        i. Comparing with batch processing: batch processing gave us exactly-once s
        executed exactly once. There are 2 parts to it: a/ dividing inputs into small

l
ut sequence of events as compared to looking at DB changes logs in CDC.
ent, you do not know what application will do with it. Also to get DB state
te, if needed)

istent but queries are forgotten after execution. In stream processing, queries
a daemon job to run on DB can be converted into to a stream processing job.

matches the query then they emit a event, say raising an alarm

erver was offline for some reason, you'll have wrong time.

We can apply this offset to each event timestamp (as seen by client).
delay.
gregate results. There are following types of windows:

time
ple is client was active for 24 hrs and then was inactive for 30mins followed
s is application dependent.
worth figuring out.

er threshold for time difference between these events to be joined. For
of these events happened in span of few minutes. Else you wont be able to

ts. DB table itself might be updated. In which get DB state updates as stream.
ther set of tables. Twitter timeline use case: after sending a tweet all
timeline of each user as a table.
s for different events. For example, if DB table was updated concurrently
vent will have a uuid for item in DB. So each event strictly maps to a specific
DB so log compaction will suffer.

emantics. Although individual jobs may fail/retry, you get output as if all was

executed exactly once. There are 2 parts to it: a/ dividing inputs into small

ii. Micro-batching and checkpointing: both are solutions to forcefully dump c stream.
    1) Micro-batching: Spark limits itself to a window of 1 sec (configurable less work to catch up on but higher co-ordination is needed and vice
    2) Checkpointing: Flink introduces barrier messages in stream which fo
iii. Idempotence and atomic transaction: Neither of micro-batching or checkp This means there can be duplicate external side effects.
    1) Atomic transaction: you can use systems like XA/2phase commit etc. protocol itself. Overhead of transaction protocol can be amortized b
    2) Idempotence: Even if operations are not idempotent, they can be ma is failing over.

er chunks b/ Idempotence of execution of each chunk
urrent state so incase of failure, we don't have to start from beginning of

e) and dumps state after every 1 sec. Smaller batches means in case of failure
versa.
rces system to checkpoint
ointing helps if system processing same event again in case of node failure.

. Since the system is homogeneous transactions can be built into the stream
y processing several input messages in single transaction.
ade idempotent by adding extra metadata. Extra care is needed when leader