Ερωτήσεις:

[1] Διαφορές μεταξύ GFS και HDFS;

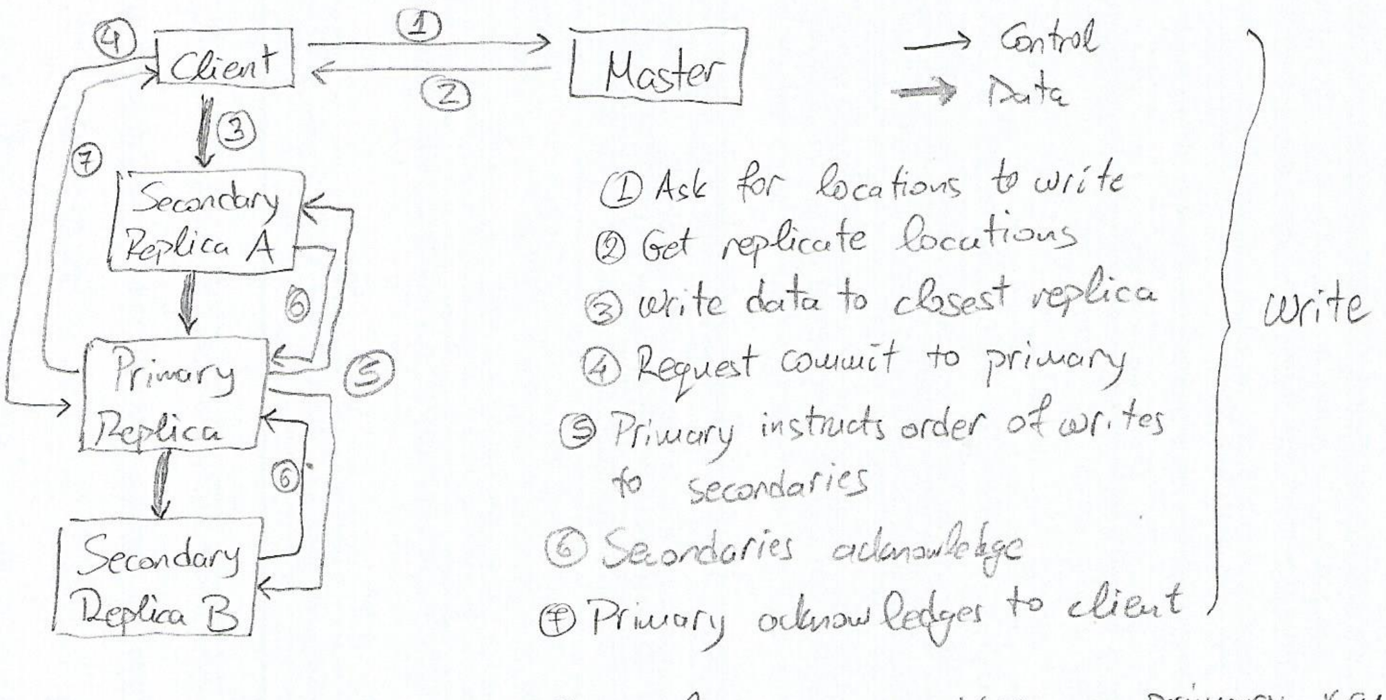
GFS: Closed Source	HDFS: Open Source
Chunk Size: 64MB	Block Size: 128MB
MasterNode - ChunkServer	NameNode - DataNode
Multiple Write - Multiple Read	Write Once - Read Many

[2] Πώς εξασφαλίζει το HDFS Fault Tolerance;

Κρατά ένα αντίγραφο στον τοπικό κόμβο, ένα σε άλλο κόμβο του rack, ένα σε απομακρυσμένο rack και επιπλέον αντίγραφα σε τυχαίους κόμβους. Πέραν αυτού, από την πλευρά του NameNode, κρατείται αρχείο των συναλλαγών όχι μόνο στον κόμβο του, αλλά και αλλού, σε περίπτωση που χρειαστεί να αντικατασταθεί.

Αντίστοιχα για υψηλή διαθεσιμότητα

[3] Πώς λειτουργεί το GFS;



① Ask for locations to write
② Get replicate locations
③ Write data to closest replica
④ Request commit to primary
⑤ Primary instructs order of writes to secondaries
⑥ Secondaries acknowledge
⑦ Primary acknowledges to client

Mutation: σε ένα αντίγραφο αποδίδεται lease, οπότε γίνεται το primary και διαχειρίζεται το mutation στα υπόλοιπα.

[4] Μια σύνοψη του BigTable;

Tablet servers έχουν tablets → σύνθεση SSTables: αποτελούνται από blocks. Τα tablets έχουν start-end (αλφαβητικά). Ο master αναθέτει tablets σε tablet servers. Το μοίρασμα γίνεται μέσω GFS. Οι ενημερώσεις γράφονται στη μνήμη στο memtable (αν μεγαλώσει το κάνουμε SSTable και φτιάχνουμε νέο). Το

[5] Πώς μπορούμε να κάνουμε βελτιώσεις στο BigTable;

- Locality groups (ομαδοποιούν column families)
- Caching (Scan cache: key-value pairs / Block cache: SSTable)
- Compression (τα δεδομένα που είναι κοντά μοιάζουν μεταξύ τους)
- Commit log (σε επίπεδο Tablet server)
- Bloom Filters: χρησιμοποιούνται για να δούμε εάν κάποιο SSTable περιέχει δεδομένα από συγκεκριμένο row, χωρίς να ανοίξουμε το row (λίγος παραπάνω αποθηκευτικός χώρος, αλλά μπορούμε να αποκλείσουμε μεγάλο αριθμό SSTables γλιτώνοντας χρόνο). Μπορεί να έχουμε False Positive, αλλά ποτέ False Negative.

[6] Ποια είναι τα βήματα του MapReduce;

(i) Πρώτα split τα input files σε M κομμάτια (16MB-64MB) και start up πολλά copies στο cluster.
(ii) Ένα copy για master, τα υπόλοιπα για workers. Ο master αναθέτει M map tasks και R reduce tasks.
(iii) Κάθε map worker parsάρει key-value pairs στην map συνάρτηση και τα αποτελέσματα (intermediate results) γίνονται buffer στη μνήμη.
(iv) Περιοδικά, τα buffered pairs γράφονται στον τοπικό δίσκο σε R περιοχές, ανάλογα με το πού τα hashάρει η partition function. Οι τοποθεσίες αυτές πηγαίνουν στο master, γιατί αυτός είναι υπεύθυνος να ενημερώσει τους reducers.
(v) Όταν ειδοποιηθεί κάθε reducer, διαβάζει και κατόπιν sortάρει όλα τα intermediate data βάσει των intermediate keys.
(vi) Κάθε reducer περνά ένα-ένα τα keys και για κάθε key εφαρμόζει τη reduce function. Τα αποτελέσματα γίνονται append σε ένα output file.
(vii) Όταν όλα ολοκληρωθούν, ο master κάνει wake up το user program.

[7] MapReduce Data Locality;

Το χαρακτηριστικό αυτό επιταχύνει τη διαδικασία, καθώς αποφεύγεται η συμφόρηση του δικτύου μέσω μεταφοράς υψηλού όγκου δεδομένων. Έτσι,

[8] Τι είναι η partition function;

Είναι η συνάρτηση που εκτελείται περιοδικά αφότου ολοκληρωθεί η διαδικασία map για να hasháρει τα intermediate keys των intermediate αποτελεσμάτων σε R τμήματα.

[9] Τι είναι οι combiners;

Είναι βελτιστοποιητές που μπορούν να εφαρμοστούν αμέσως μετά τη map, προκειμένου να κάνουν κάποιο aggregation by intermediate key (συνήθως τη reduce function, γι' αυτό συχνά καλείται και mini-reduce) πριν το shuffling.

[10] Τι είναι τα MapReduce shuffle & sort;

Shuffle: διαμοιρασμός των intermediate results στις τοποθεσίες των reducers (ίδια keys στον ίδιο reducer)

Sort: το sorting που πραγματοποιεί κάθε reducer πριν την εφαρμογή της reduce, αφού μπορεί να έχει λάβει πολλά keys

[11] MapReduce: small ή large dataset;

Το MapReduce ενδείκνυται για batch processing, επομένως το potential του αξιοποιείται σε πολύ μεγάλα datasets. Σε μικρότερα, οι διαδικασίες συγχρονισμού ενδέχεται να εξισορροπήσουν τα προτερήματα της παραλληλοποίησης. Επίσης, εάν το dataset είναι τόσο μικρό που να χωρά στη μνήμη, η εξοικονόμηση χρόνου είναι 1 τάξης μεγέθους, αφού δε χρειάζονται τα writes on disk που γίνονται μετά τις maps

[12] Τι συμβαίνει όταν ένας mapper αποτυγχάνει καθώς στέλνονται τα δεδομένα του σ' έναν reducer;

Εάν δεν υπάρχουν αντίγραφα επανεκτελείται, αλλιώς speculative execution (αυτό βέβαια αφορά κυρίως γενικά να μην περιμένουν όλοι τον πιο αργό worker, αφού πολλοί κάνουν το ίδιο).

[13] Πλήθος mappers / reducers;

[illegible] Total size

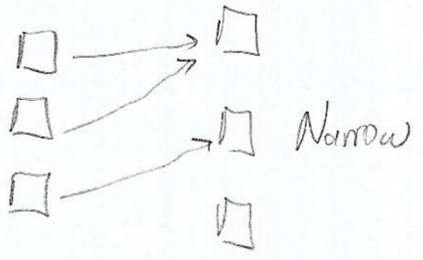
[14] Lazy Transformations;

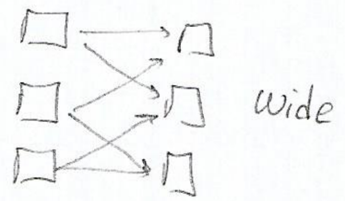
Αντί κάθε intermediate μετασχηματισμός να εφαρμόζεται απευθείας στα δεδομένα, τα αποτελέσματα να συλλέγονται και η διαδικασία να επαναλαμβάνεται εκ νέου, η πληροφορία του μετ/μου οδηγεί στη δημιουργία ενός νέου RDD, το οποίο φέρει πληροφορία για το lineage του. Οι μετ/μοι πραγματοποιούνται όμως μόνο όταν δοθεί κάποια οδηγία τύπου execution (π.χ. collect), δίνοντας έτσι ως τότε τη δυνατότητα βελτιστοποίησης ως προς τους μετασχηματισμούς.

[15] Narrow - Wide dependencies;

Narrow: κάθε τμήμα ενός parent RDD αντιστοιχεί σε ένα μόνο τμήμα (το πολύ) του child RDD. (εύκολο, θέλει λίγους workers)

Wide: ένα τμήμα ενός parent RDD μπορεί να αντιστοιχεί σε περισσότερα από 1 τμήματα του child RDD. (αργό, θέλει shuffle & sort)





[16] Sort or Hash shuffle;

Hash: κάθε mapper δημιουργεί ένα αρχείο για κάθε reducer, οπότε τελικά έχουμε M·R αρχεία. Είναι γρήγορο και δε χάνεται χρόνος I/O, όμως δε συμφέρει για πολύ μεγάλο αριθμό τμημάτων.

Sort: εξάγεται 1 μόνο indexed αρχείο με δεδομένα ταξινομημένα ανά reducer id. Προφανώς είναι πιο αργό από το hashing, αλλά αν δεν έχουμε μικρό αριθμό τμημάτων ή SSD, συμφέρει.

Default: Sort shuffle, εκτός εάν έχουμε λίγους reducers

[17] Reduce side join;

Γνωστό και ως **shuffle join**. Mappers αναλαμβάνουν κάθε dataset και κάνουν emit ως intermediate results tuples με key το κλειδί συνένωσης και value τα υπόλοιπα στοιχεία. Το πλαίσιο συγκεντρώνει τα tuples, αναθέτει σε κάθε reducer tuples με κοινό κλειδί συνένωσης

... δεδομένου πως δεν υπάρχει εξασφάλιση ως προς τη σειρά με την οποία γίνεται το join (πρώτα R και μετά S ή ανάποδα;). Πιθανές λύσεις σε αυτό είναι [1] να κρατιέται στη μνήμη αυτό που θέλουμε να μπει πρώτο (συμφόρηση — η μνήμη μπορεί να κληθεί να κρατήσει πολλά τέτοια) ή [2] να δημιουργούνται πιο σύνθετα intermediate keys, χρησιμοποιώντας τα values.

[18] Map side join;

Γνωστό και ως sort-merge join. Προϋπόθεση είναι τα δεδομένα να είναι ταξινομημένα ως προς το κλειδί συνένωσης και το partitioning τους να έχει γίνει με τον ίδιο τρόπο σε ίσα μέρη (συν-διαχωρισμός).
⇒ Είναι ρεαλιστικό να περιμένουμε κάτι τέτοιο;
Σε πολλές περιπτώσεις ναι, διότι το join μπορεί να κληθεί να γίνει στα πλαίσια ενός ευρύτερου workflow, οπότε να έχουν εκπληρωθεί οι συνθήκες λόγω προηγούμενων διεργασιών.

[19] Hash join;

Γνωστό και ως broadcast join. Προϋπόθεση είναι το ένα dataset να είναι σημαντικά μικρότερο από το άλλο, ώστε να φορτωθεί στη μνήμη σε ένα hashmap με κλειδί το κλειδί συνένωσης. Το άλλο dataset αναγιγνώσκεται και γίνεται probe για το κλειδί συνένωσης. (Το μικρό διαμοιράζεται σε όλους τους mappers, το μεγάλο σπάει, κάθε mapper κάνει probe του κλειδιού συνένωσης στο μικρό)

<u>Σημείωση</u>: Στα [18], [19] δεν απαιτούνται εν γένει reducers.

[20] Τι join συμφέρει;

Σε ταχύτητα ισχύει hash > map-side > reduce-side.
Αλλά πάνε ανάποδα ως προς το πόσο general purpose είναι.

[21] Τι συμβάλλει στο κόστος ενός SQL ερωτήματος;

I/O μεταφορές μεταξύ δίσκου και μνήμης, φόρτος στην επικοινωνία
[illegible]

[22] Query Optimization;

(Thanks gramaria) Δεδομένου ενός λογικού πλάνου των επιθυμητών υλοποιήσεων, προσπαθούμε να πραγματοποιήσουμε βελτιστοποιήσεις ως εξής: μεταφέρουμε selection και projection (filters) operations όσο πιο πίσω γίνεται στο pipeline, ώστε τα joins να μείνουν «για το τέλος» και να πραγματοποιηθούν σε όσο το δυνατό μικρότερο τμήμα των datasets. Πρόσθετα, επιλέγουμε το είδος του join που ενδείκνυται δεδομένων των συνθηκών.

Σχετικό ερώτημα περί βελτιστοποίησης

[23] Columnar vs Row-oriented databases;

Αφορούν σχεσιακές βάσεις δεδομένων. Οι row-oriented είναι εσωτερικά σχεδιασμένες για να είναι βελτιστοποιημένες στη διαχείριση γραμμών. Ενδείκνυνται για workloads τύπου OLTP και γενικά όπου χρειάζεται πληροφορία (read/write) για πολλές ή και όλες τις στήλες. Αν μόνο μερικές από τις στήλες μας απασχολούν, τότε χρησιμοποιούμε columnar, οι οποίες είναι εσωτερικά σχεδιασμένες για να είναι βελτιστοποιημένες στη διαχείριση στηλών. Παραδείγματα file formats που έχουν columnar indexing είναι τα ORC of Morgoth και το Parquet.