ADBS 2020

Part I - Optimizing relational queries

Storage

Storage hierarchy: primary (transient; CPU registers, caches, main memory), secondary (persistent; magnetic disks, flash memory, solid-state drives) and tertiary storage (removable; tape, CD, DVD)

Memory close to CPU fast, small and expensive; memory at periphery slower, larger and less expensive

Magnetic disks: disks rotate steadily and are managed in blocks, system reads/writes data one block at a time => data can only be read/written when head is positioned over it; access time $t_a = t_s + t_r + t_{tr}$ with the following components:

- t_s seek time: move disk arm to desired track (usually given)
- t_r rotational delay: disk controller waits for desired block to rotate under disk head => usually ($\frac{0.5}{rpm}$) minutes (0.5 is the average value from 0 to 1)
- t_{tr} transfer time: read/write data (number of bytes divided by the transfer rate, mind the conversions!)

If blocks from different tracks are read the track-to-track seek time t_{t2t} is also relevant, since it describes the time that the arm requires to move arm to next track

SSDs: neither seek time nor rotational delay => only t_{tr} of relevance

Record blocking: records must be allocated to disk blocks => block of size B and fixed-length record of size R has blocking factor $bfr = \lfloor B/R \rfloor$ => unused space in each block: $B - (bfr * R)$ bytes

Spanned block organization stores part of record in one block and stores a pointer to block where record is continued

Indexing

Primary index: file and index ordered on the ordering key field (unique values), file contains full records, index contains key value + pointer; data access via primary index faster since binary search on index faster than on the larger data records

Clustering index: file and index ordered on the ordering NON-key field (may have non-unique values), otherwise same behaviour as primary index

Secondary index: index is ordered on the ordering field (may be unique or not), file is either ordered on another field or unordered

B^+**-tree** usually default index method, B-tree hardly used; B^+ -trees are always balanced, never require overflow pages, log-time search/insert/delete and all nodes except root have at least 50% occupancy

Hash-based indexes: allocation of N "buckets", use hash function to map each possible value to one of the buckets $[1, N-1]$, each bucket has pointer to chain of overflow pages; hash index unbeatable for equality-based queries, performance degrades with number of overflow pages, useless for range queries; solutions to overflow pages are static or extendible hashing

Bitmap indexes: well-suited for set operations /logical AND/OR), counting of rows and search with OR and NOT conditions

Query processing basics

Rule-based optimization: apply equivalence-preserving transformations to make intermediate results as small as possible (e.g. replacing cross product and subsequent selection with join)

"Canonical translation" of SELECT-FROM-WHERE query:

1. Cartesian product of all relations from FROM-clause
2. Selection with predicate from WHERE-clause
3. Projection to attributes in SELECT-clause

Typical transformations on such a canonical translation would be:

- Cascading selections
- Pushing selections (apply selections as early as possible)
- Replace Cartesian product + selection by join
- Project out attributes that are not needed further up in the tree, i.e.: not contained in query result or in select/join predicate
- Join order: carry out most selective selections/joins first
- Further step: identify groups of operations that can be executed together by a single algorithm (in particular: select, project)

Cost-based optimization: find the query plan (query execution strategy) with minimal estimated cost (or a plan with reasonably low estimated cost in case the search space is too big) => "cost" is some metric that measures the required resource (e.g. disk I/O, CPU cost, memory usage cost, communication cost)

Evaluation of relation operators

Join implementations: nested loops join, block nested loops join, index nested loops join, sort-merge join, (hybrid) hash join

Cardinality estimation

Histograms: equi-depth (number of rows in each bucket is roughly the same) vs equi-width (all intervals have equal size)

Selectivity: ratio of qualifying vs all tuples

Part II - Hadoop & MapReduce

MapReduce

Challenges of distributed data processing: synchronization of processes and data, finite bandwidth => communication may become new bottleneck, partial failures (e.g. 1000 servers, mean team to failure ~3years => ~1 failures per day on average) => restarting not an option, system must support partial failures

Distributed file system (DFS): huge files (100s of GB/TB), rarely updated, mostly reads and appends, divided into 64/128MB chunks which are typically replicated 3 times on different racks; master/name node contains metadata (= chunk positions) for file and may itself be replicated

Pioneering work at Google (Google File System), open source realization (Apache software foundation) => Hadoop distributed file system (HDFS)

MapReduce: map function takes a chunk of data from the DFS and outputs a sequence of key/value pairs, reduce function takes a key plus the associated list of values and produces a sequence of key/value pairs (which can serve as input to another map or be the final output)

Execution of MapRedcude job:

- Map tasks: Each map task gets split(s) from the DFS and runs user-defined map function on each record of a split
- Shuffle and sort: Key/value pairs produced by map tasks are sorted and grouped by key, the values associated with each key are formed into a list
- Reduce tasks: reducer applies the reduce function to a value list of 1 key and each reduce task may be assigned several keys

Combiner functions may be used to push some reducer logic to the map task => reduces the communication cost (e.g. calculate sum of word occurrences in a document for each mapper node in the word count example)

Single reducer (no parallelism, may take long, nodes may run out of disk space with lots of intermediate data) vs one reduce task per key (maximum parallelism, computation overhead) => good number of reducers determined by expected amount of intermediate data (typically use fewer reduce tasks and nodes than key values so longer tasks can run on one node while multiple shorter ones run on another)

Partitioner can be used to determine which reduce task receives which keys and associated value lists (partitioners usually apply hash functions so pairs with same key go to same reducer, custom partitioners for better load balancing can be implemented if user has domain knowledge)

Complexity aspects of MapReduce algorithms:

- total communication cost: total number of I/O actions, only sum up input sizes
- replication rate: average number of key/value pairs that mappers create from each input
- reducer size: maximum size of value list of one key (in case of skew there might be huge discrepancies between the length of these value lists)
- wall clock time: time to execute the parallel algorithm
- processing time (usually ignored since it is much smaller than the communication cost)

Hadoop

Open source platform/framework for big data processing, contributors include Yahoo, IBM, Facebook

Hadoop ecosystem comprises HDFS, MapReduce, Hive (SQL interface over MapReduce), Spark (in-memory processing engine), HBase (NoSQL database, column family) and many more components

HDFS: based on ideas of the Google File System (GFS), on top of native file system (runs in user space, files have to be copied from local file system to HDFS and back), provides redundant storage for massive amounts of data, performs best with smaller number of larger files (millions of files each with 100MB+), optimized for large and streaming reads of files (rather than random reads) and is typically written once and read often

HDFS splits files into blocks (typically 64/128MB), which simplifies storage and replication, and distributes blocks at load time (typically 3 copies)

Nodes in HDFS are either name/master nodes (maintain file system tree and contain metadata for all files) or data/worker nodes (store and retrieve data blocks)

Hive's goal is to provide SQL-like interface for data in Hadoop => user formulates queries in Hive-QL (based on SQL) and system transforms queries into jobs of underlying execution engine (e.g. MapReduce); enables large-scale data analysis w/o need of deep SE knowledge, more productive than writing MapReduce directly => 5 lines of Hive-QL may correspond to 100 lines of Java

Hive's basic principle is **schema-on-read:** you can store data in HDFS w/o knowing a particular format, knowledge of schema only required when you want to analyze data => flexible and interoperable, but conflicts between expected/actual data are not detected during writing

Spark

Fast and general-purpose cluster computing platform, extends MapReduce model, able to run

computations in memory, well-suited for various tasks in the data analysis pipeline

Spark stack contains multiple, closely integrated components

Every Spark application consists of a driver program that launches various parallel operations on a cluster => access of a driver program to Spark happens through SparkContext object, which represents a connection to the cluster

Part III - NoSQL

Distributed database management systems

Distributed databases are a connection of database nodes over a computer network, logical interrelation of the connected databases, possible absence of homogeneity among connected compute nodes

Distributed DBMS, a software system that manages a distributed database (vs centralized DBMS running on single node) have many advantages: increased availability (avoid single point of failure, isolate faults to their site of origin, ability to replace a failed node), improved performance through load balancing and locality of data, flexible scalability since servers can join and leave the network, support for heterogeneous nodes since nodes can be upgrade in steps

Main architectures of DDBMS: shared memory (access same main memory, additional CPUs may even slow down system because of memory contention), shared disk (multiple CPUs with own memory, access to common disks, similar problems as with shared memory), shared nothing (avoids bottlenecks and may achieve linear speedup and linear scale-up)

Fragmentation: breaking a relation into smaller relations (=fragments) and storing the fragments individually (maybe at different sites); fragmentation approaches: horizontal fragmentation (also sharding; subset of the tuples in that relation), derived horizontal fragmentation (apply partitioning of primary relation to secondary relation via foreign key), vertical fragmentation (divide relation vertically by columns, each fragment holds only certain attributes of the relation => PK or some unique key attribute in every fragment otherwise reconstruction is not possible) and mixed (hybrid) fragmentation (arbitrary combination of horizontal and vertical fragmentation, reconstruction becomes complex)

Requirements of fragmentation are completeness (original dataset must be reconstructable) and non-redundancy/disjointness (any two fragments share no attributes other than PK)

Granularity of fragmentation very important (neither too coarse-grained e.g. entire relations nor too fine-grained since reconstruction becomes very expensive)

Allocation: distributing the fragments over available sites with or without replication

Replication: storing several replicas (=copies) of a fragment at different sites, number of copies per data item is called replication factor

Availability (failing replicas can be covered by other nodes) and performance (of reads especially, since load balancing, data locality and parallelization enable read from any replica) are two advantages of replication; downsides are consistency (all replicas have to be updated if data item is updated) and concurrency (different users update multiple replicas at the same time)

Master-slave replication (write requests are handled by a single dedicated server, which is responsible for updating all other servers that hold replicas => avoids concurrency problem, but master may become bottleneck) vs multi-master replication (every server holding a replica of a data record may process write requests, regular synchronization between servers but higher write availability & parallel processing of write requests)

Goals of fragmentation/allocation/replication are data locality (data frequently accessed together should be in same fragment), minimization of communication costs, improved efficiency (queries on smaller fragments typically faster than on large data sets), load balancing

Distribution transparency DDBMS should appear to user like centralized DBMS, different aspects:

- Location transparency: user command independent of location of data and where command is entered
- Fragmentation/replication transparency: hide fragmentation and replication from user
- Migration transparency: data items may be moved from one server to another without affecting user
- Concurrency transparency: DDBMS must be able to handle multi-user access (much more complex than for single server DBMS)
- Failure transparency: failures are more likely than on a single server; DDBMS should do its best to continue processing user requests in case of failures
- Execution transparency: user does not need to know where a transaction is executed

Failures in distributed systems:

- Server failure: server crash, delay, erroneous behavior
- Link failure: communication link between two servers unable to correctly transmit messages
- Message failure: loss or delay of messages, unintended duplication
- Network partition: network split into multiple subnets that cannot communicate with each other

Availability refers to the probability that the system is operational at a given time

Reliability refers to the probability that the system operates as specified (=failure-free) at a given time

Fault-tolerance is the ability to detect failures and minimize consequences (e.g. no inconsistent data or minimization of downtime), it recognizes that failures may happen; important special case of fault-tolerance is partition tolerance, the ability to continue operating while a network is split

Consider the following aspects: access types, access patterns (frequency), affinity of records, access durations

Distributed concurrency control and recovery

Distributed concurrency control: how to maintain locks for replicated objects, deadlock detection; distinguished copy approach, voting approach

Distributed recovery: ensure atomicity of distributed transactions, cope with new failure types (failure of some but not all sites, failure of communication links), synchronization of the data after recovery before re-joining the network

Distributed deadlock can happen even if no local graph contains cycle (e.g. T2 waits for T1 at site A, T1 waits for T2 at site B => global wait graph contains deadlock), can be detected in centralized way (each site sends waits-for-graph to one site which does deadlock detection), in hierarchical way (idea: deadlocks most likely on related sites => nodes construct local waits-for-graph in its subtree of hierarchy, periodically send its graph to its parent node in the hierarchy in increasing intervals) or by time-out (abort transaction if it waits longer than chosen interval => very simple since cooperation between sites is not required, may cause unnecessary aborts)

Specific to DDBMS => **phantom deadlock:** delay in propagating local information may lead to false positives of deadlock detection algorithm (e.g. deadlock of T1,T2 at sites A and B, T2 may be aborted due to error at site C => if this message arrives late then deadlock detection may cause abort of T1)

Two-Phase commit (2PC): coordinator (transaction manager at the site where transaction originates) and subordinates (transaction managers at sites where some subtransactions are being executed) involved in 2PC, 2 phases (voting and termination phase), both initiated by coordinator, coordinator and subordinate can unilaterally abort a transaction

Problems of 2PC: failure of coordinator (transaction blocked until coordinator recovers, subordinates cannot decide status of transaction even if they have connection and everyone voted YES, since coordinator could still vote NO) => 3PC proposed (but not used in practice due to significant additional communication cost)

Distributed query processing

Stages of distributed query processing: query mapping (supplied query refers to global schema), localization (map distributed query to separate queries on individual fragments at different sites), global query optimization (select optimal execution plan according to cost factors => disk I/O, CPU, memory, communication costs of input data, intermediate results and transfer of final result to user's site), local query optimization (apply usually query optimization techniques locally)

NoSQL systems

Motivation: Relational DBMS have strong formal foundation, standard data model and query language, concurrency (support multi-user access and transactions), data integration (store data of many applications in one database), but also have many limitations (schema must be known before any data can be added, designed for structured data, requires horizontal & vertical homogeneity of data, impedance mismatch => relational model vs in-memory data structures)

New data management challenges: processing "big data", complex data structures, schema independence & evolution, sparseness (many data items unknown or non-existent => many NULL values in relational tables)

Not only SQL (NoSQL): umbrella term for DB systems that are open-source, distributed, non-relational, schemaless, restricted querying capabilities, horizontally scalable, run well on clusters, no (or weaker form of) transactions, BASE instead of ACID

Benefits of NoSQL database systems: high performance, horizontal scalability, schemaless, no (or weaker form of) transactions

Cons of NoSQL database systems: no standardized query language, weaker form (or no support of) transactions, limited support of ensuring integrity

4 categories of NoSQL systems: key-value stores (e.g. Redis), document stores (e.g. MongoDB), column databases / column family stores (e.g. Cassandra), graph databases (e.g. Neo4J)

Application vs integration database: shift idea of integration DB to application DB => each app maintains its own DB, data exchange via web services, rich structure for communication to reduce number of round trips, more freedom in data modeling

Use of cluster: scale out to cope with huge amounts of data, RDBMS not well-suited for running in clusters

ACID/BASE: atomicity, consistency, isolation, durability; basically available, soft state, eventually consistent

CAP properties: consistency, availability, partition tolerance => CAP theorem states that in cluster one can only have 2/3 of the properties

PACELC extends CAP by covering the A-C trade-off in case of network partition, else trade-off between latency and consistency (4 types: PA/EC, PA/EL, PC/EC, PC/EL)

Quorums are defined to provide a flexible mechanism, i.e. balancing the performance of reading and writing, when ensuring consistency on a cluster, differentiation between read quorum R (# of replicas that have to be contacted for read operation to succeed) and write quorum W (# of replicas that have to be contacted for write operation to succeed)

Quorum requirements (assuming replication factor N):

- Overlap of two writes $W + W > N$: more than half of the replicas are reached by write operation
- Overlap of write-read $R + W > N$: at least one node accessed by the read was involved in last write

Quorum overlap guarantees read requests to see the most recent version of the data item (requires version stamps) and prevents concurrent writes since more than half of the nodes have to accept the write

Quorum extreme cases: ROWA (read-one-write-all: $W = N, R = 1$) and majority quorum ($W > \lfloor N/2 \rfloor + 1$ and $R > \lfloor N/2 \rfloor + 1$

Timestamps simple but not reliable in distributed environment, synchronization of clocks in different nodes is complex => logical clocks are a counter for messages/events, clock ticks are represented by an increment of the counter, counter values are compared to decide order of versions

Scalar (=Lamport) clocks: send message happens before receiving it, take transitive closure; notation $e_1 \rightarrow e_2$ (i.e. e_1 happened before e_2)

Scalar clock initialization sets counter C_i on every server i to 0, before server i sends message the counter C_i is incremented and attached to the message, when server j receives the a message with counter C_i the set $C_j := max(C_i, C_j) + 1$. Global clock $C$of events is updated each time the counter is updated on a node, i.e. $C(e) := C_i(e)$. Scalar clocks provide no total order (two messages can have same counter). Lamport clocks satisfy weak clock property (not string clock property due to the independent sequences of events on different nodes).

Weak clock property: for any two events e_1 and e_2: if $e_1 \rightarrow e_2$ then $C(e_1) < C(e_2)$

Strong clock property: or any two events e_1 and e_2: $e_1 \rightarrow e_2$ iff $C(e_1) < C(e_2)$

Vector clocks: every node maintains vector of clocks for all participating nodes, $VC_i[j] =$ clock of node j in vector at node i. Initialization sets $VC_i[j]$ to 0 for every i and j. Before server i sends message counter $VC_i[i]$ is incremented and VC_i is attached to the message. When server j receives message from node i with vector VC_i then do:

- For every k, $VC_j[k] := max(VC_i[k], VC_j[k])$
- Set $VC_j[j] := VC_j[j] + 1$

Vector clocks satisfy the string clock property, order on vector clocks as follows:

- $VC \leq VC'$ iff $VC[k] \leq VC'[k]$ for every k
- $VC < VC'$ iff $VC \leq VC'$ and there exists at least one k s.t. $VC[k] < VC'[k]$

Key-value stores

Stores values as byte arrays indexed by keys, limted collection of data types (e.g. text, JSON, XML, binary), semantics of the value only known to application, no complex queries (mostly look-up by key), focus on scalability, performance and high availability => pioneered by Amazon's DynamoDB

Key naturally available and used for data access (e.g. sessionID => session data, date => log file, login/email => user attributes)

Riak: open source implementation following Amazon's Dynamo, key/value store with some extras, distributed and horizontally scalable, highly available and fault-tolerant, built for the web, basic operations: GET, PUT, DELETE, extras: secondary indexes, links to other objects, full-text search, MapReduce

Document stores

Similar to key-value stores but DBMS can understand format of the value (typically data is either XML, JSON, BSON), allows advanced querying and relationships (references) between data objects, joins usually not supported due to performance (=> denormalizing data when modelling or combining documents on application level), atomicity usually only guaranteed on documents

MongoDB: open-source, stores data in BSON, use cases include log data, e-commerce, inventory/content management, master-slave distribution

MongoDB principles: keep intended data access in mind, aim for atomic reads and writes if possible, denormalization is an option, respect the maximum document size of 16MB, use document validation, decide which indexes shall be created

Graph databases

Convenient way to store entities (as nodes) and relationships between them (as edges), edges are directed (if relationship holds in both directions both edges have to be inserted), optimized for graph traversal queries, relationships are first-class citizens

Property graph model: graph DB consists of nodes and named relationships, nodes can be tagged by labels to express their role, both nodes and edges can have properties (list of key/value pairs), analogy with language: nodes => nouns, named relationships => verbs, whiteboard-friendliness (data model stays essentially the same as drawn on the whiteboard)

Graph DB use cases: NOT only social networks, real time recommendations, master data management, banking/finance

Benefits: intuitiveness of data model, speed (of development and execution, index-free adjacency allows for efficient traversal of large DBs since edges to/from adjacent nodes are stored like pointers), agility (easy to adapt to changes)

Neo4J: core servers (masters) and read replicas (slaves), fully ACID compliant, asynchronous propagation of data modifications to the replicas (= eventual consistency), "read your writes" consistency guaranteed by Neo4J (causal consistency)

Graph DBs not well-suited for sharding (where to split) => application level sharding or cache sharding

Column stores

Business Analytics & Data Warehouses usually involve aggregation over few columns => with column stores only required columns are read (much less I/O)

Explicit (associate a tuple identifier to every entry in a column; more data on disk but required sometimes) vs implicit IDs (use offset in column as virtual ID, problems with compressed data and variable-width columns)

"Pure" column stores, column-orientated extensions for other systems and wide column stores

Vectorized processing: columns stores can make use of SIMD (single instruction multiple data) operations, also better cache locality compared to iterating over rows, even without SIMD

Compression: analytical queries require a lot of I/O and aggregation (cheap for CPU) => trade CPU time for I/O by storing data compressed (e.g. run-length encoding, bit-vector encoding, dictionary encoding, NULL suppression)

Late materialization combines the columns as late as possible so column-oriented optimizations can be exploited as much as possible

Wide column store: no agreed upon definition, schmealess and inherently sparse columnsm designed as highly distributed systems, also called column family stores (attributes usually accessed in groups => column families)