Aside: **CAP** Theorem

• For any shared-data system...

– **Consistency**: all nodes have the same copies of a replicated data item visible for various transactions (note: different from ACID)

– **Availability**: every request (to a non-failed node) will result in a response

– **Partition tolerance**: system still operates despite inability of nodes to communicate

Common NoSQL Categories (1)

• **Document Stores** – Commonly JSON, no schema

– Access by document id or content – Example: MongoDB

• **Key-Value Stores** – Associative array: access data via id – Basically fast, distributed hash tables – Example: DynamoDB, Redis

• **Graph** – Stores nodes, columns; graph ops optimized – Example: Neo4j

• **Others** – Column-based (horizontal partitioning) – Hybrid – XML – Objects

Attributes in Queries

–  Queried = potentially good for indexes

–  Updated = bad for indexes

–  Unique = could be indexed

**In Index Ordering of columns is VERY important**

Functionality

An index answers certain kinds of questions very efficiently (depends upon type of index)

– Equality: fieldname=value

– Range/ordering: fieldname>value

***• Only index that maintains ordering (e.g. tree-based)***

***Can be used for WHERE clause, as well as JOIN and ORDER BY***

Index on (C,A), (C,B), ... (i.e. start with C) Anything not starting with C = **full table scan**  \

Index on (A, B)

A = 'cat' (fast)

A = 'cat' AND B >= 3 (fast, if ordered)

A <= 'panda' ORDER BY B (fast, if ordered)

Anything not starting with A = **full table scan**

Index on (B) B = 1 (fast)  B <= 5 (fast, if ordered) ORDER BY B (fast, if ordered)

**T1 JOIN T2 ON T1.B=T2.Y**

• No indexes: scan T1, scan T2 (n2)

• Index on T1(B): scan T2, fast search in T1

• Index on T2(Y): scan T1, fast search in T2

• Index on T1(B), T2(Y): ***merge sort (if ordered)***

**Pro**

• Can make the difference between full table scan and log/constant lookup

**Con**

• Extra space

– Linear with # rows

• Extra time

– Creation (moderate)

– Maintenance (can offset savings) \

Choosing the Index(es) to Create

Table size

– Many rows = larger cost to table scan

Data distribution (selectivity)

– Fewer distinct values = higher likelihood needing to touch many rows, independent of index usage

• Index can lead to lots of IO/cache misses vs. sequential scan via clustered index

Query vs. update load

– Many updates = higher relative index maintenance cost

– Analysis of frequent queries leads to choosing key attributes that get you the most bang for your buck

• Cardinality: # distinct values in a column

– SELECT COUNT (DISTINCT col\_name) FROM table\_name;

• **Selectivity: 100% \* cardinality / # rows**

– Compare for 10K rows...

• Gender (M/F)

• Country (195 + Taiwan)

• Birthday (Jan. 1 -> Dec. 31)

-Use ***narrow indexes*** (i.e. few columns); these are more efficient than compound indices

-***Avoid a large number*** of indices on a table

-***Avoid “overlapping”*** indices that contain shared columns  (often a single index can service multiple queries)

-For indices that contain more than one column: given no other constraints, place the ***most selective column first***

-Unless you have very good reason, ***always define a PK*** (in most RDBMSs, results in a clustered index, more shortly)

***Index Types***

***• Clustered vs. Non-clustered***

***• Covering (w.r.t. a query)***

***• Balanced Trees (B+-Trees)***

***• Hash Tables***

***• Other***

Clustered vs. Non-clustered

• ***Clustered: affects physical order on disk***

– At most one per table (for some RDBMSs, PK) – Fast when data accessed in order/reverse

• ***Non-clustered: induces logical ordering***

– Arbitrary number per table

– Depending on the query/data, can lead to significant slowdown due to cache misses and frequent disk access

***Covering index***

Typically indexes help the DBMS find the row of interest  – ID -> Name – Name->ID

A covering index contains all the necessary data within the index itself (w.r.t. to query or queries)  – More storage vs. IO savings – (ID, Name) or (Name, ID)

***B+-Trees***

Balanced, constant out-degree (within range)

Values (i.e. row pointer) only at leaves – Distinguishes from a B-tree – Linked list at leaves, in order

Logarithmic traversal, constant at leaf – Top k levels usually kept in memory (e.g. 2-3)

Typical default index for DBMS; also used in file systems, etc.

***Other***

• Bitmap – Useful for low-update systems (e.g. read-only) with low

cardinality attributes (e.g. gender) • Trie

– Useful for sequence queries (e.g. bioinformatics) • Spatial (e.g. R-tree)

– Useful for queries about space (e.g. what stores are close to me? what planes are within 1 mile of each other?)

• Inverted – Useful for full-text search (e.g. search engines)

**Denormalization**

The goal of normalization is to yield a database schema that is free from redundancies

Depending upon performance constraints and the job mix, sometimes it is appropriate to introduce redundancies (i.e. denormalize to 1/2NF) in the name of performance improvement (e.g. to avoid joins)

NOTE: a schema should always be fully normalized first, and denormalization considered during physical tuning upon analysis of constraints/performance  – This technique should be deliberate and is not an excuse for sloppy database design

**Common Denormalization Uses**

1. Storing derived attributes  – Every iPhone has a list of prior owners, each with a name and e-mail. The price of the device depends upon how many prior owners there have been.
2. Adding attributes to a relation from another relation with which it will be joined  – Profiling has shown us that every query on employee project assignments has needed the project name.
3. Storing results of calculations on one or more fields within the same relation  – We need to store chemicals in base units (e.g. mL), but our most frequent query depends upon larger units (e.g. L)

**Database Design Tuning**

1.Denormalization is one method by which to alter database design to achieve performance goals

2.Others common approaches...

– Vertical partitioning

– Horizontal partitioning

**Vertical Partitioning**

- Given a normalized relation [typically with many attributes], break into two or more relations, each duplicating the PK, but separating attribute groups

- Example: • GivenR(K,A,B,C,G,H,...)

– Knowing that (A,B,C) typically together, distinct from (G, H,...)

• YieldR1(K,A,B,C)andR2(K,G,H,...)

**Horizontal Partitioning**

Given a normalized relation [typically with many rows], break into two or more relations, each with the same columns, but a different subset of rows

Example:

– Given ORDER(ID,REGION\_ID,...) • Knowing that typical queries are specific to a region

– Yield ORDER\_R1(ID,...), ORDER\_R2(ID,...), ...

• Will require multiple queries/UNION if all orders are to be considered at once

**Query Issues (1) INDEXES NOT USED**

Many query optimizers do not use indexes in the presence of...

1. ***Arithmetic expressions*** – Salary/2000 > 10.50
2. Numerical comparisons of attributes of ***different sizes and precision*** – Aqty=Bqty,whereAqtyisINTEGERandBqtyisSMALLINTEGER
3. ***NULL comparisons*** – ReportsTo IS NULL
4. ***Substring comparisons*** – Lname LIKE '%mann'
5. Indexes are often not used for ***nested queries*** using IN:

*SELECT Ssn FROM EMPLOYEE*

*WHERE Dno IN ( SELECT Dnumber FROM DEPARTMENT*

*WHERE Mgr\_ssn = '333445555' );*

The DBMS may not use the index on Dno in EMPLOYEE, whereas using Dno=Dnumber in the WHERE-clause with a single block query may cause the index to be used.

1. Some DISTINCTs may be redundant and can be avoided without changing the result.

***A DISTINCT often causes a sort operation and must be avoided as much as possible***

1. Avoid ***correlated queries*** where possible.

Consider the following query, which retrieves the highest paid employee in each department:

SELECT Ssn

FROM EMPLOYEE E

WHERE Salary = (SELECT MAX(Salary)

FROM EMPLOYEE M WHERE M.Dno=E.Dno);

This has the potential danger of searching all of the inner EMPLOYEE table M for each tuple from the outer EMPLOYEE table E

To make the execution more efficient, the process can be re-written such that one query computes the maximum salary in each department and then is joined

1. If multiple options for a join condition are possible, choose one that ***avoids string comparisons***

For example, assuming that the Name attribute is a candidate key in EMPLOYEE and

STUDENT, it is better to use EMPLOYEE.Ssn = STUDENT.Ssn as a join condition rather than EMPLOYEE.Name = STUDENT.Name

1. One idiosyncrasy with some query optimizers is that the ***order of tables in the FROM***-clause may affect the join processing.

If that is the case, one may have to switch this order so that the smaller of the two relations is scanned and the larger relation is used with an appropriate index.

Some DBMSs have commands by which to influence query optimization (e.g. HINT)

1. A query with multiple selection conditions that are ***connected via OR*** may not be prompting the query optimizer to use any index. Such a query may be split up and expressed as a union of queries, each with a condition on an attribute that causes an index to be used. For example,

SELECT Fname, Lname, Salary, Age FROM EMPLOYEE WHERE Age > 45 OR Salary < 50000;

may be executed using table scan giving poor performance. Splitting it up as

SELECT Fname, Lname, Salary, Age FROM EMPLOYEE WHERE Age>45 UNION SELECT Fname, Lname, Salary, Age FROM EMPLOYEE WHERE Salary < 50000;

*may utilize indexes on Age as well as on Salary*

1918489