“Beyond SQL“

php|tek 2006 in Orlando Florida

Lukas Kahwe Smith
smith@pooteeweet.org

Agenda:

- The “SQL“ Standard
- Understanding Performance
- Normalization
- Tables and Columns
- Indexes
- Joins and Subqueries
- Locks

The "SQL" Standard: History

- Standard [English] Query Language
- Standard language to talk to Relational Database Management Systems (RDBMS)
- Pronounced [SEQUEL]
- ANSI Standard
  - 1986 (SQL 87)
  - 1989 (SQL 89)
  - 1992 (SQL 92)
  - 1999 (SQL3)
  - 2003 (SQL:2003)

The "SQL" Standard: Issues

- Does not cover all behavioral aspects
  - SQL actually does not cover a lot of things that people think are part of the standard!
- Is not free of ambiguity
- Often followed, rather than led, vendor implementation
- Not all vendors chose the same ways to implement the standard
  - Do not expect things to work the same on every database!

Understanding Performance: Benchmarking

- Set of isolated performance test cases
- Indicator for how an application would perform if it were to use the given code
- Repeat with disabled caching
- Change one parameter at a time
- Store results for later reference
- Dangerous: understand all aspects of benchmark before making conclusions!
- Tools: Super Smack, ApacheBench, etc.

Understanding Performance: Profiling

- Method of diagnosing the performance bottlenecks of a specific application
- Pin point trouble spots that you may then isolate, benchmark and tweak
- Spend the most time on areas where your application spends the most time
- Benchmark advice also applies
- Tools: EXPLAIN and other DBMS tools, user land profiler (like APD)

Understanding Performance: Explain

- Show execution plan for a given query
  - How will the table(s) be scanned?
  - What indexes will be used?
  - What join algorithms will be used?
  - What is the estimated „execution cost“?
- Tool of choice for query optimizations
- Not part of the SQL standard
- All DBMS have some equivalent
  - SET EXPLAIN, SELECT .. PLAN, etc.

Understanding Performance: Optimizers

- Rule-based optimizers use non volatile data and fixed assumptions
- Cost-based optimizers additionally use table statistics and other volatile data
  - Everybody claims to be cost-based
  - Biggest advantage for cost-based optimizers is for joins
- Statistics may change over time
  - Use ANALYZE, OPTIMIZE, VACUUM or some other RDBMS specific command to keep tables in mint condition

Normalization: Overview

- Process of removing redundant data
- Normalization helps avoid INSERT UPDATE and DELETE anomalies
- Anomalies cause
  - INSERT to require unnecessary data
  - DELETE remove too much data
  - UPDATE to deal with redundant data
- Multiple normal forms (NF) exist
- Most databases are third normal form
- There are even stricter normal forms

Normalization: Example

- PRIMARY KEY uniquely identifies a row
- CANDIDATE KEY is any set of columns that uniquely identify a row
- Example non normalized table
  - Name is the PRIMARY KEY
  - Salary could be considered a CANDIDATE KEY for this set of data

Diplomats					
Name	Title	Language	Salary	Workgroup	Head
Axworthy	Consul	French, German	30,000.00	WHO, IMF	Greene, Craig
Broadbent	Diplomat	Russian, Greek	25,000.00	IMF, FTA	Craig, Crandall
Craig	Ambassador	Greek, Russian	65,000.00	IMF	Craig
Crandall	Ambassador	French	55,000.00	FTA	Crandall
Greene	Ambassador	Spanish, Italian	70,000.00	WHO	Greene

Normalization: First Normal Form - 1NF

- All columns only contain scalar values as opposed to lists of values
  - Split Language, Workgroup, Head
  - Name, Language and Work_group now compromise the PRIMARY KEY

Diplomats

Name	Title	**Language**	Salary	**Workgroup**	Head_honcho
Axworthy	Consul	French	30,000.00	WHO	Greene
Axworthy	Consul	German	30,000.00	IMF	Craig
Broadbent	Diplomat	Russian	25,000.00	IMF	Craig
Broadbent	Diplomat	Greek	25,000.00	FTA	Crandall
Craig	Ambassador	Greek	65,000.00	IMF	Craig
Craig	Ambassador	Russian	65,000.00	IMF	Craig
Crandall	Ambassador	French	55,000.00	FTA	Crandall
Greene	Ambassador	Spanish	70,000.00	WHO	Greene
Greene	Ambassador	Italian	70,000.00	WHO	Greene

Normalization: Dependence

- A Column is set dependent if its values are limited by a another column
- A column is functionally dependent if
  - For every possible value in the the column
  - There is one and only one possible value set for the items in a second column
  - Must hold true for all possible values
- Column A is transitively dependent on column C if
  - Column A is dependent on column B which in turn is dependent on column C

Normalization: Second Normal Form - 2NF

- All nonkey columns must be functionally dependent on the primary key
  - Title, Salary are not functionally dependent on the Language column
  - Head is set dependent on Workgroup

Diplomats

Name	Title	Salary	**Workgroup**	Head
Axworthy	Consul	30,000.00	WHO	Greene
Axworthy	Consul	30,000.00	IMF	Craig
Broadbent	Diplomat	25,000.00	IMF	Craig
Broadbent	Diplomat	25,000.00	FTA	Crandall
Craig	Ambassador	65,000.00	IMF	Craig
Crandall	Ambassador	55,000.00	FTA	Crandall
Greene	Ambassador	70,000.00	WHO	Greene

Languages

Name	**Language**
Axworthy	French
Axworthy	German
Broadbent	Russian
Broadbent	Greek
Craig	Greek
Craig	Russian
Crandall	French
Greene	Spanish
Greene	Italian

Normalization: Third Normal Form - 3NF

- All nonkey columns must directly dependent on the primary key
  - Head is only dependent on the Name through the Workgroup column

Diplomats

Name	Title	Salary
Axworthy	Consul	30,000.00
Broadbent	Diplomat	25,000.00
Craig	Ambassador	65,000.00
Crandall	Ambassador	55,000.00
Greene	Ambassador	70,000.00

Work_groups

Workgroup	Head
WHO	Greene
IMF	Craig
FTA	Crandall

Affiliations

Name	**Workgroup**
Axworthy	WHO
Axworthy	IMF
Broadbent	IMF
Broadbent	FTA
Craig	IMF
Crandall	FTA
Greene	WHO

Languages

Name	**Language**
Axworthy	French
Axworthy	German
Broadbent	Russian
Broadbent	Greek
Craig	Greek
Craig	Russian
Crandall	French
Greene	Spanish
Greene	Italian

Normalization: When not to Normalize

- Intentionally violating normalization rules is sometimes feasible
  - Improve performance
  - Simplify queries
- Application needs to handle anomalies
- RDBMS provide different features to automatically handle “denormalization“
  - Indexes
  - Materialized Views
  - Triggers

Normalization: When to Normalize

- Always normalize unless you are concerned with a single bottleneck
- Expect performance reduction for other cases when denormalizing
- Normalized tables are smaller and therefore allow for faster retrieval
- Instead of denormalize optimize joins

Tables and Columns: Storage Hierarchy

Database

Tablespaces

Files

Extents

Pages

Tables and Columns: General Tips

- Minimal unit of I/O is a page (not a row)
- I/O = Disk I/O + CPU I/O / 1000 + Net I/O * 1.5
- Page size should be same as cluster size
- Reading multiple rows from a single page has a constant cost
- Use PCTFREE or FILLFACTOR to prevent shifts for expanding UPDATE's

Tables and Columns: Character Columns

- CHAR, VARCHAR and NCHAR (VARYING)
- Variable length columns save space and accurately handle trailing whitespace
  - For sorts defined, not real, length matters!
  - Size storage overhead for variable length columns costs between one and four bytes
- Fixed length reduce risk of page shifts

Tables and Columns: Character Sorts

- Binary sort
  - Fastest sort
  - Somewhat non intuitive code page based
  - Case sensitive
- Dictionary sort
  - Requires conversion step
  - Dictionary like sorting
- Dictionary sort with tie breaking
  - Also sort by accents and letter case
- Pick the sort type at table creation or use COLLATE (or CAST) at runtime

Tables and Columns: Numerical Columns

- INTEGER, FLOAT, DECIMAL, SERIAL
- When choosing size beware of overflow danger on arithmetic operations
- Consider using character types for columns that require character function
- FLOATs may use CPU floating point unit

Tables and Columns: Temporal Columns

- DATE, TIME and TIMESTAMP
- All fixed length
- Again try to be consistent in the type choice to prevent casts
- Many DBMS internally always use TIMESTAMP for storage
  - No additional space requirement
  - May affect sort speed however

Tables and Columns: LOBs

- BLOB, CLOB and NCLOB
- Use when character type does not provide sufficient space
- Usually stored on a separate page from the rest of the row
  - Reduce number of pages if data in LOB column is seldom accessed
  - Changes in LOB do not cause page shifts
  - When not move LOB into it's own table
- Do not allow many of the common character functions or set functions

Tables and Columns: Sorting Speed

- Number of columns in the ORDER BY
- Length of the columns in the ORDER BY
- Number of rows
- Partial duplicates hurt performance
- Presorted sort faster than random sets
  - Do not assume that sorting is instant if the data is already in order!
- DBMS will try to keep data in memory if the records in the ORDER BY are small
- INTEGER beat SMALLINT beat CHAR

Indexes: General Tips

- Critical for performance is the number of layers in the btree not the size
  - Rebuild if 5% of all rows have changed
  - Prefer non volatile columns for indexing
  - Use bitmap indexes when selectivity is low
  - Clustered indexes cause rows to be stored in order of the clustered key
- Some DBMS multiple indexes per query
- Some DBMS do not include NULLs
- Most RDBMS allow forcing index use
  - Do not force index reading 20%+ rows

Indexes: Covering and Compound Indexes

- DBMS will use a covering index to fetch data instead of the table when possible
- Are not used in joins or groupings
- Put the most used and most selective column first
  - Index on (A, B, C) implies indexes on (A) and (A, B)
- Put columns in the query in the same order as in the index

Joins and Subqueries: Nested Loop Joins

```
for (each page in outer_table) {
  for (each page in inner_table) {
    for (each row in outer_table_page) {
      for (each row in inner_table_page) {
        if (join column matches) {
          pass;
        } else {
          fail;
        }
}}}}
```

Joins and Subqueries: Nested Loop Joins

- Conclusion is DBMS will make the
  - Smaller table the inner table
  - Table with a good index the inner table
  - Table with more restrictive/expensive WHERE clause outer table
- For multi column joins add a matching compound index on the inner table
- Use the same data type and size in both tables for the columns in the join
- Use irrelevant restrictions on one table to force it to become the outer table

Joins and Subqueries: Nested Loop Joins

Joins and Subqueries: Sort Merge Joins

```
sort (t1); sort (t2); // <- expensive
get first row (t1); get first row (t2);
while (rows in tables) {
  if (join-col in t1 < join-col in t2)
    get next row (t1);
  elseif (join-col in t1 > join-col in t2)
    get next row (t2);
  elseif (join-col in t1 = join-col in t2)
    pass;
    get next row (t1); get next row (t2);
}
```

Joins and Subqueries: Sort Merge Joins

- Advantage
  - one pass reading instead of multi pass with nested-loop joins
- Disadvantage
  - cost of sorting, requires more RAM, startup overhead
- Perfect if you have a clustered key on the join columns in both DBMS

Joins and Subqueries: Hash Joins

- Hash join is a nested loop join where a hash is computed for the inner table
- Useful mainly as a fall back from nested loop joins and sort merge joins
  - No restrictions on large outer table
  - Not a lot of RAM to spare
  - Data is not presorted
  - No indexes

Joins and Subqueries: Join Advantages over Subquery

- Optimizer has more choices
  - subquery forces a nested-loop algorithm
- Multiple WHERE clauses in the outer table can be reordered easier in a join
- Some DBMS can parallelize joins better
- It is possible to have columns from both tables in the select list
- Due to their greater popularity they are used more and therefore optimized more in DBMS

Joins and Subqueries: Subquery Advantages over Join

- ANY or EXISTS can break out early
- Column type mismatches are less costly
- Only recently more DBMS are getting the ability to join in UPDATE
  - MySQL limits subqueries in DML
- They read more easily as they are „modular“

Locks: Introduction

- Shared locks: reading
  - N shared locks may coexist
- Update locks: reading + planned update
  - N shared locks may coexist with one update lock
- Exclusive locks: writing
  - One exclusive lock may not coexist with any other lock
- Granularity may be database, table, page, row (and a few others)
- Lock granularity may get escalated up

Locks: Isolation Levels

- READ UNCOMMITED
  - no locks
- READ COMMITED (common default)
  - may release lock before transaction end
- REPEATABLE READ (common default)
  - may not release lock before transaction end
- SERIALIZEABLE
  - concurrent transactions behave as of executed in sequence

Locks: Multi Version Concurrency Control

- Keep copy of modified data around until all transactions have ended that started before the change occurred
  - PostGreSQL appends (use VACUUM)
  - Oracle, MySQL, Interbase/Firebird overwrite
- Effectively evades locking readers
- Emulate using optimistic locking
  - Locks at commit time
  - Add unique “transaction id“ to row id
  - Reads will work but UPDATEs will fail if someone else has changed the data

Locks: Deadlocks

- Deadlock is when multiple transactions wait for one another to release locks
  - Use READ ONLY and FOR UPDATE
  - Escalate locks early in the transaction with dummy UPDATE or LOCK statements
  - Access tables in the same order in all transactions
  - Split transactions up as much as possible
  - Do validation and computation in the client before starting a transaction

References:

- These slides
  - http://pooteeweet.org/files/phptek06/beyond_SQL.pdf
- „SQL Performance Tuning“
  - by Peter Gulutzan and Trudy Pelzer
- Normalization
  - http://dev.mysql.com/tech-resources/articles/intro-to-normalization.html
- Images in the database
  - http://mysqldump.azundris.com/archives/36-Serving-Images
  - http://mysqldump.azundris.com/archives/37-Serving-Images-from-a-File-System.html
- Discussions over MVCC
  - http://www.ibphoenix.com/main.nfs?page=ibp_mvcc_roman

Thank you for listening ..
Comments? Questions?

smith@pooteeweet.org