“fast, portable, SQL II”
international php conference 2005

Lukas Kahwe Smith
smith@pooteeweet.org
Agenda:

- Joins and Subqueries
- Indexes and Constraints
- Data changes
- Locks
- Optimizer
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 to make the
  - Smaller table the inner table
  - Table with a good index the inner table
  - Table with more restrictive/expensive WHERE clause outer table
- DBMS will use these principles to decide the join strategy
- If you think you know better put unnecessary restrictions on the table you want as the outer table
Joins and Subqueries: Nested Loop Joins

- 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
- Encourage use of index order to reduce disk head jumping
  - SELECT * FROM t1, t2
    WHERE t1.col1 = t2.col2
    AND t1.col1 > 0
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);
}
Join 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
Hash joins and beyond:

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

- Use join indexes, composite tables or materialized views
- Remember constant propagation
  - `SELECT * FROM t1, t2
    WHERE t1.col1 = t2.col2
    AND t1.col1 = 42`
- May transform favorably to
  - `SELECT * FROM t1, t2
    WHERE t1.col1 = 42
    AND t1.col2 = 42`
Joins and Subqueries: ANSI vs. Old Style Joins

- **ANSI style**
  - SELECT * FROM t1 JOIN t2 ON t1.col1 = t2.col1

- **Old style**
  - SELECT * FROM t1, t2 WHERE t1.col1 = t2.col1

- Both are equally fast
- But obviously make use of ANSI style to replace old hacks with set operators to "emulate" outer joins
Joins and Subqueries: Join Advantages over Subquery

- Optimizer has more choices
  - subquery forces a nested-loop
- 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

- One (outer table) to many (inner table) relations benefit from the ability to break out early
- Column type mismatches are less costly
- Only recently more DBMS are getting the ability to join in UPDATE
- They read more easily as they are “modular”
Joins and Subqueries: Subquery Optimizations

- Subqueries work in-to-out or out-to-in
- Add DISTINCT to query inside the IN to get rid of duplicates and to get presort
- Transform UNION to double IN query
- Transform NOT IN to EXCEPT
- Transform double IN subqueries to the same table to a “row subquery”
- Merge multiple subqueries into one
- Replace “> ALL” with “> ANY” with a MAX() in the subquery to use index
Indexes and Constraints: General Tips

• Critical for performance is the number of layers in the btree not the size
  – Rebuild if a number of rows equivalent 5% of all rows was added or changed
  – Prefer non volatile columns for indexing
  – Use bitmap indexes for large, static data
  – Clustered indexes cause rows to be stored in order of the clustered key

• Recent DBMS versions increasingly are able to use multiple indexes per query

• Most RDBMS allow forcing index use
Indexes and Constraints: Compound Indexes

- Put up to 5 columns in compound index
- Put the most used and most selective column first
  - This implies a single column index on the first column in the compound index
- Put columns in the query in the same order as in the index
Indexes and Constraints: Covering Indexes

• DBMS will use a covering index to fetch data instead of the table when possible
• Are not used in joins or groupings
• If you do not care about NULL and the name column is indexed
  – SELECT name FROM t1 ORDER BY name
• May transform favorably to
  – SELECT name FROM t1
    WHERE name > '' ORDER BY name
Indexes and Constraints: Constraints

- Define a FOREIGN KEY constraint with the same data type, column size and name as the PRIMARY KEY it references.
- PRIMARY KEY and UNIQUE usually imply an index but FOREIGN KEY does not.
  - Do not create redundant indexes.
- Optimizer will use additional information about uniqueness or storage order.
- TRIGGER are expensive, syntax varies, react only to explicit data changes.
**Data Changes: INSERT**

- Make use of DEFAULT values where possible to cut down on network traffic
- Put primary or unique columns first
- Make use of bulk inserting syntax
- Consider disabling CONTRAINTS during bulk changes
- Update your statistics during low traffic
Data Changes: UPDATE

- Put most likely to fail SET clause left
- Add a redundant WHERE to speed things up when no change is required
  - UPDATE t1 SET col1 = 1
- May transform favorably to
  - UPDATE t1 SET col1 = 1 WHERE col1 <> 1
- Use CASE to merge two UPDATE statements on the same column
- Consider moving columns to one table if they need to updated in sequence often
Data Changes: DELETE

- Use TRUNCATE (or DROP TABLE) to remove all rows from a table
- Put DELETE before UPDATE and INSERT to reduce risk of page shifts
- Or more generally put shrinking data changes before expanding data changes
**Data Changes: Transactions**

- Set auto commit on for single statement transactions
- Put the statements that are likely to fail at the start of the transaction, especially if the ROLLBACK will likely be expensive
- ROLLBACK is usually more expensive than COMMIT
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 UNCOMMITTED**
  - no locks
- **READ COMMITTED (common default)**
  - may release lock before transaction end
- **REPEATABLE READ (common default)**
  - may not release lock before transaction end
- **SERIALIZABLE**
  - concurrent transactions behave as if executed in sequence
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 statement
  - 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
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
- This 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
Retrieving Data II: Rule-based vs. Cost-based

- 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
References:

- These slides
  - http://pooteewet.org/files/phpconf05/fast_portable_SQL_II.pdf
- „SQL Performance Tuning“ by Peter Gulutzan and Trudy Pelzer
- Benchmarking and Profiling
- SQL Syntax Tips
  - http://troels.arvin.dk/db/DBMS/
Thank you for listening..

Comments? Questions?

smith@pooteewleet.org