

“fast, portable, SQL I”

international php conference 2005

Lukas Kahwe Smith
smith@pooeteeweeet.org

Agenda:

- The “SQL” Standard
 - Understanding Performance
 - Tables and Columns
 - Simple Searches
 - Sorting and Aggregation
-
-

The “SQL“ Standard

- 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

- 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.
-
-

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)
 - $I/O = \frac{\text{Disk I/O}}{1000} + \frac{\text{CPU I/O}}{1000} + \text{Net I/O} * 1.5$
 - 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: General Tips

- Some DBMS support ROLLBACK on DDL statements others do implicit COMMIT
 - Put NULL-able and variable length columns at the end of the table
 - Be consistent to prevent type casting
-
-

Tables and Columns: Normalization

- 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
 - Optimize your joins instead of denormalizing
-
-

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: 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
 - Timezone handling is up to you
-
-

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 functions
 - Prefer DECIMAL over FLOAT
 - Emulate SERIAL with a trigger
 - Oracle is all different: only one variable length type exists internally
-
-

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
-
-

Simple Searches: Code Points

Operator	Points	Operand	Points
=	10	Literal alone	10
>	5	Column alone	5
>=	5	Parameter alone	5
<	5	Multiop. Expression	3
<=	5	Exact numeric type	2
LIKE	3	Other numeric type	1
<>	0	Temporal type	1
		Character type	0
		NULL	0

Simple Searches: Code Points Examples

- WHERE some_char = 'The answer: 42!'
 - 0 Points for „character type“ on the left
 - 5 Points for „column alone“ on the left
 - 10 Points for „equal“ operator
 - 10 Points for „literal alone“ on the right
 - 25 Points Total

Simple Searches:

Code Points Examples (continued)

- WHERE some_int <= another_int + 27
 - 2 Points for „exact numeric type“ on the left
 - 5 Points for „column alone“ on the left
 - 5 Points for „smaller or equal“ operator
 - 3 Points for „multi operand expression“
 - 2 Points for „exact numeric type“ on the right
 - 17 Points Total

Simple Searches:

Constant Propagation

- Fold constants where possible
 - WHERE col1 = 12 AND col1 = col2
 - My transform favorably to
 - WHERE col1 = 12 AND col2 = 12
 - Beware of NOT
 - WHERE column1 = 12 AND NOT (column3 = 17 OR column1 = column2)
 - May transform favorably to
 - WHERE column 1 = 12 AND column3 <> 17 AND column2 <> 12
-
-

Simple Searches: Dead Code Elimination

- Eliminate „always false“ conditions
 - WHERE 0 = 1 AND col1 = 'my comment'
- Though sometimes its useful to force an index lookup on a non NULL column
 - WHERE some_indexed_column > 0



Simple Searches: Constant Folding

- DBMS often do not fold obvious constant folding cases
 - WHERE some_column + 0
 - WHERE some_column IN (3, 42, 42)
 - WHERE CAST(1 AS INTEGER)
 - WHERE a - 4 = 17
 - May transform favorably to
 - WHERE some_column
 - WHERE some_column IN (3, 42)
 - WHERE 1
 - WHERE a = 21
-
-

Simple Searches:

Case-Insensitive Searches

- Not all DBMS let you choose a COLLATION during the run time
 - Use LOWER() over UPPER() in order to not lose information
 - UPPER('Voilà') != 'voilà'
 - LOWER('Voilà') = 'voilà'
 - Portability trick looking for sensible combinations to skip LOWER() usage
 - WHERE column1 = 'LASTNAME' OR 'Lastname'
-
-

Simple Searches: Search Condition Optimal Form

- $\langle \text{column} \rangle \langle \text{operator} \rangle \langle \text{literal} \rangle$
 - WHERE col1 -12 = col2
 - May transform favorably to
 - WHERE col1 = col2 +12
 - Column left, simple value right
 - „standard“ 32bit INTEGER are „simpler“ than 16bit SMALLINT
 - WHERE some_smallint * some_smallint
 - Is slower on some DBMS/CPU's than
 - WHERE some_int * some_int
-
-

Simple Searches: AND Ordering

- Most DBMS evaluate each component of a search condition from left to right
 - Put least likely first
 - Put least complex first
 - Rule based (versus cost based) optimizers might re-order things again
 - we will get to that topic later



Simple Searches: OR Ordering

- Remember last page
 - Put most likely expression left
 - Put least complex expression left
 - Put same columns together
 - WHERE col1 =< 10 OR col2 = 2 OR col1 = 20
 - May transform favorably to
 - WHERE col1 = 20 OR col1 =< 10 OR col2 = 2

Simple Searches: AND Plus OR

- Make use of the distributive law to reduce comparisons
 - WHERE (A AND B) OR (A AND C)
- May transform favorably to
 - WHERE A AND (B OR C)



Simple searches: NOT

- Have a tendency to obscure readability
 - ... WHERE NOT (column1 > 5)
 - Is harder to read than
 - ... WHERE column1 <= 5
 - Do not transform a NOT on an indexes column to unequal as unequal values outnumber equal values in an even set
 - But if you are looking for a rare value
 - WHERE NOT (bloodtype = 'O')
 - May transform favorably to
 - WHERE bloodtype > 'O' OR bloodtype < 'O'
-
-

Simple Searches:

IN

- Favor IN over a series of OR's on the same column
 - WHERE col1 = 7 OR col1 = 9
- May transform favorably to
 - WHERE col1 IN (7, 9)
- If possible make use of BETWEEN
 - WHERE col1 BETWEEN 7 AND 9

Simple searches:

LIKE

- Favor equal over LIKE where possible
 - watch out for trailing whitespace
 - WHERE column1 LIKE 'some string'
- May transform favorably to
 - WHERE column1 = 'some string'
- Replace SUBSTRING() with LIKE

Simple searches: SIMILAR, MATCHES and REGEXP

- Pattern matching capabilities
- Expect fairly different syntax
- Replace OR logic with pattern
 - WHERE column1 IN ('A', 'B', 'XX')
- May transform favorably to
 - WHERE column1 SIMILAR TO '[AB]|XX'



Simple searches: Set Operator

- Merge set operators where possible
 - `SELECT * FROM t1 WHERE c1 = 2 UNION
SELECT * FROM t1 WHERE c2 = 4`
 - May transform favorably to
 - `SELECT DISTINCT * FROM t1
WHERE c1 = 2 OR c2 = 4`
 - Only exception are older DBMS that do not support using multiple indexes when both columns are indexed
-
-

Simple searches:

CASE

- Use CASE to cut down on multiple function calls with the same parameter
 - WHERE some_slow_func() = 'foo'
OR some_slow_func() = 'bar'
- May transform favorably to
 - WHERE CASE some_slow_func()
WHEN 'foo' THEN 1
WHEN 'bar' THEN 1 END

Simple Searches: LIMIT, TOP, FETCH FIRST

- Cuts down on network traffic
 - Not included in the SQL standard
 - Emulate in the client, with CURSORS or with a subquery using ROW_NUMBER()
 - ```
SELECT * FROM (
 SELECT ROW_NUMBER() OVER
 (ORDER BY key ASC) AS rownum,
 columns FROM tablename) AS foo
WHERE rownum > skip
 AND rownum <= (n+skip)
```
- 
-

# *Simple Searches: Consistent Style*

- Maintain a consistent style in your SQL to assist binary comparisons in query caches
  - Use the same case
  - Use the same order
  - Use the same whitespace formatting



# *Sorting and Aggregation: General Considerations*

- Factors that affect 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 sets usually 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
- 
-

# *Sorting and Aggregation: Data Types*

- INTEGER sort faster than SMALLINT
- INTEGER sort faster than CHAR
- Sets (no duplicates) beat multi-sets
- Conclusion
  - ascending sort of an unique presorted INTEGER column is the fastest



# *Sorting and Aggregation: NULL Sorting*

- Some DBMS sort NULL high and some low (interbase puts them at the end)
  - {NULL, 0, 1} vs. {0, 1, NULL}
- Implementation defined in SQL 99
- SQL 2003 adds NULLS FIRST|LAST



# *Sorting and Aggregation: Implicit Ordering*

- Some SQL constructs have a side effect of sorting data on most DBMS
    - Clustered keys
    - Searches on indexed columns
    - DISTINCT on non unique columns
  - In those cases adding an ORDER BY may degrade performance
  - However you cannot hold the DBMS „liable“ to behave this way
- 
-

# *Sorting and Aggregation: 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
- 
-

# Sorting and Aggregation: Encouraging Index Use

- Force index usage by inserting a „redundant“ search condition
    - `SELECT * FROM t1 ORDER BY c1`
  - May transform favorably to
    - `SELECT * FROM t1  
WHERE c1 >= '' ORDER BY c1`
  - Not really „redundant“ because it eliminates NULL values
  - If you are feeling adventurous remove the ORDER BY to rely on index sorting
- 
-



# *Sorting and Aggregation: Presorting*

- Use clustered key
- Export and reimport ordered
- Add a rough key INTEGER column for the first part a wide column
  - ORDER BY rough\_key, wide\_column
  - Remember you can represent characters as integers



# *Sorting and Aggregation: Optimal GROUP BY Clause*

- Removing redundant columns in the GROUP BY clause
  - Most DBMS only allow to have columns in the select list that are in the GROUP BY except inside aggregate functions
    - `SELECT col2, primary_col, COUNT(*)  
FROM t1 GROUP BY col2, primary_col`
  - May transform favorably to
    - `SELECT MIN(col2), primary_col, COUNT(*)  
FROM t1 GROUP BY primary_col`
- 
-

# Sorting and Aggregation: Avoid JOINS

- GROUP BY reduces row count while a JOIN increases row count
  - Consider set operators like UNION, EXCEPT, INTERSECT to replace joins
  - Consider subquery if t1.c1 is unique
    - SELECT COUNT(\*) FROM t1, t2  
WHERE t1.c1 = t2.c1
  - May transform favorably to
    - SELECT COUNT(\*) FROM t2  
WHERE t2.c1 IN (SELECT t1.c1 from t1)
- 
-

# Sorting and Aggregation: Redundant HAVING Clauses

- Remove conditions in the HAVING clause that are already ensured by the WHERE clause in an SQL query
  - `SELECT c1 FROM t1 WHERE c2 = 2  
GROUP BY c1 HAVING c1 > 6`
  - May transform favorably to
    - `SELECT c1 FROM t1 WHERE c2 = 2  
AND c1 > 6 GROUP BY c1`
  - Unless condition is very expensive to evaluate
- 
-

# *Sorting and Aggregation: Avoid GROUP BY*

- Consider using DISTINCT instead of GROUP BY if no set functions are used as its simpler, allows expressions and is faster on some DBMS
  - SELECT col1 FROM t1 GROUP BY col1
- May transform favorably to
  - SELECT DISTINCT col1 FROM t1



# *Sorting and Aggregation: ORDER BY and GROUP BY*

- Consider putting in matching GROUP BY and ORDER BY clauses
  - GROUP BY col1, col2 ORDER BY col1
- May transform favorably to
  - GROUP BY col1, col2 ORDER BY col1, col2



# Sorting and Aggregation: Indexes

- Used in simple GROUP BY statements
    - `SELECT col1 FROM t1 GROUP BY col1`
  - Consider UNION for queries with MIN() and MAX() call on an indexed column
  - Encourage index use
    - `SELECT MIN(col) FROM t1;`
  - May transform favorably to
    - `SELECT col FROM t1 ORDER BY col LIMIT 1;`
  - No need on PGSQL 8.1 and MYSQL 5
  - `COUNT(col1)` might use covering index
- 
-

# *Sorting and Aggregation: CUBE and ROLLUP*

- Few DBMS fully support CUBE and ROLLUP to get detail and summary report information at the same time
- Emulate using a UNION





# *Sorting and Aggregation: Trust Your Query Cache*

- Obvious temptation is to denormalize data that is expensive to fetch
- Remember that frequently accessed information might be in the query cache



# References:

- These slides
    - [http://pooteeweet.org/files/phpconf05/fast\\_portable\\_SQL\\_I.pdf](http://pooteeweet.org/files/phpconf05/fast_portable_SQL_I.pdf)
  - „SQL Performance Tuning“ by Peter Gulutzan and Trudy Pelzer
  - Benchmarking and Profiling
    - <http://dev.mysql.com/tech-resources/articles/pro-mysql-ch6.pdf>
  - SQL Syntax Tips
    - <http://troels.arvin.dk/db/DBMS/>
- 
-

Thank you for listening ..  
Comments? Questions?

[smith@poteeweet.org](mailto:smith@poteeweet.org)

---

---