CS631 Implementation Techniques for Relational Database Systems

List of Group Projects

Instructions: Projects are marked as P1, P2, etc. In implementing these, first do overall specs and the design. Work out your approach, identify postgreSQL modules to be modified/extended/replaced. Each project requires that you study the relevant techniques (most discussed in the class, or consult suggested papers). This will be good place to meet me or any TA and get your approach validated before you start coding. Plan your testing and test data also carefully.

Materialized view definition and maintenance

Modify the code for defining views to also create an actual table. Also modify code to insert/ delete/ update tuples to also do view maintenance periodically.

P1. Create select-project-join type views and show working with a view on join of 2 tables. You need to collect delta-R from log (create one if required). Use incremental view maintenance.

P2. As above but define views with aggregation on a single table only. Use incremental view maintenance.

P3. As above but use triggers or rules in postgreSQL to do view maintenance as part of same update transaction (do for select-project-join type views as well as aggregation on a single table only).

Query optimization with materialized views

P4.: in your implementation, assume that a view defined in the catalogue is a materialized view. A query on materialized view may be executed against the view or against the base table. Extend optimizer to allow this and to choose whichever is better in cost.

P5. Same as above, but extend optimizer to do the reverse where a given query is on base tables, but we can instead use the materialized table.

For P4 and P5 : read the Oracle paper sent to you. Look also at Surajit Chaudhuri and Kyuseok Shim's paper on how to do this
(http://ieeexplore.ieee.org/iel2/3041/8637/00380392.pdf?tp=&arnumber=380392&isnumber=8637)

Database Replication

P6. Publish-subscribe replication in PostgreSQL Implement a mechanism whereby an instance of postgreSQL can publish certain relations, and other instances can subscribe to it. In practical terms, the publisher keeps a list of subscribers and what they are subscribed to. Whenever an update happens on a subscribed relation, an update is sent to the subscriber. You can assume that the update can be sent over a socket connection (which the subscriber should be listening for), or even over http (using a servlet at the subcriber).

P7. Replication using log file : as above except that one instance sends the log file to the other instance periodically. Study the log format of postgreSQL, and use it to update the other instance. Ensure that logs are authentic, and same log is not processed multiple times. Also, explain and analyze what would happen if the second instance fails during recovery itself. (Note that replication is not transactional. To find out when a relation is updated, see the tips in the materialized view maintenance project. Note that updates by uncommitted transactions may be sent as a result. Extra credits for ensuring that only committed updates are sent to the subscriber)

Tip: Slony-I is replication software designed for postgreSQL. It is not full featured compared to pub-sub support in commercial databases, but take a look at it to see what features it supports. (You don't have to implement all of them!)

.

Index Selection

P8. Index selection (on which attribute to create indexes) is a hard problem, but here you need to do only a part of the job. First figure out how to create hypothetical indices (i.e. indices that are not actually present, but you fool the optimizer into thinking they are there). Now given a query, or a set of queries, you can find out the estimated benefit of adding an index. Write simple heuristics to look at a query and decide what indices may be useful. Then write a simple heuristic for choosing good indices (a greedy heuristic is a good option). You may store any extra information you need in separate tables in the catalogue.

Nested Queries:

P9. Query decorrelation for PGSQL:
Many nested subqueries can be transformed to queries with joins/outerjoins. Read Galindo-Legaria/Joshi paper (Cesar A. Galindo-Legaria, Milind Joshi: Orthogonal Optimization of Subqueries and Aggregation. SIGMOD Conference 2001.) and implement their decorrelation technique in PostgreSQL. You will have to understand how PostgreSQL parses and represents queries and then add a stage just after parsing to perform decorrelation. Decorrelation can be done always, don't worry about whether it will provide a performance benefit.

P10. Implement the pivot operation : Given, for example, a table R(A, B, C), where (A,B) forms a key for R, the operation pivot[A][B](R) gives a table with attributes (A, V1, V2, .., Vn), where V1..Vn are the distinct values that occur in column B. The tuples in the result contain for each A value, a value Ci in column Vi, if tuple (A, Vi, Ci) occurs in R. In general, we can have multiple columns in place of A, but B and C can be assumed to be single columns. Such an operation can give a table such as

```
Year Q1 Q2 Q3 Q4
------------------------
2005 11 22 15 25
2006 10 20 18 27
```

when we apply pivot[Year][Quarter](sales) on a relation sales of the form

```
Year Quarter Sales
2005 Q1 11
2005 Q2 22
2005 Q3 15
```

Allow Pivot to be part of a SQL query.

P11. Array Index: Although B-tree indices are quite efficient, they cannot beat the speed of array indexing. For example, suppose you wish to find the seats available in a particular train on particular day, from a relation avail (train, date, seats). If the data is organized as a 2-d array, with train numbers mapped to integers (using a separate lookup array) and dates similarly mapped to integers (assume only 60 days ahead are required), lookup will be very fast using arrays. This project is to build such an index, for lookups on two columns (at least, if you can make it more general, that will be great), to find the value of another column (more generally, multiple other columns). Extend the indexing mechanisms of PostgreSQL to implement such an index. When data is updated, the index must also be updated. For the purpose of this project, don't worry about (a) index concurrency control or (b) recovery/logging. In fact you can simply build the index in memory, with nothing on disk, but make sure the index gets updated (and preferably, recreated if the database is restarted). Show how this index can be integrated with the optimizer, and get used in query plans, if time permits. If you are out of time, simply show a direct lookup on this index.

P12. Progress estimator for SQL queries : Have you ever waited for a long SQL query to complete, and had no idea how much more time it would take to complete. There are a few recent papers on how to estimate how much more time a query execution will take. Implement (parts of) one of them on the postgreSQL query evaluation engine. If you handle just fully pipelined queries, that should be sufficient.
References:

* Surajit Chaudhuri, Vivek R. Narasayya, Ravishankar Ramamurthy: Estimating Progress of Long Running SQL Queries. SIGMOD Conference, 2004
* Gang Luo, Jeffrey F. Naughton, Curt Ellmann, Michael Watzke: Toward a Progress Indicator for Database Queries. SIGMOD Conference 2004, 791-802

P13. Table Partitioning : Provide ability to create partitioned tables for large tables. Assume that R is very large. We can store it as 'horizontal' non-overlapping fragments R1, R2, etc., and associate each Ri with some predicate., like

R1 : A <= 10
R2 : A > 10 and <= 20
R3 : A > 20

Naturally, R is a union of R1, R2 and R3.
Provide for partitioning large tables like this, and translating a query on R onto a query on one/more of its fragments. Develop logic which includes only the required partition in the query.

P14. Allow temporal DB extension: The table R, if defined as Temporal table, will automatically include attributes From, To for storing start and end time during which the tuple is valid. For example, when you add Ramesh as an employee with salary 15000 at time 12, the tuple inserted will be

<Ramesh, 15000, 12, forever>

(Note : 'forever is simply some special value to indicate thet the tuple is still valid). When at time 28, you change his salary to 18000, the above tuple is modified and a new tuple ias added as follows;

<Ramesh, 15000, 12, 27>
<Ramesh, 18000, 28, forever>

Represent 'forever' suitably. Allow indexes to be created on time intervals. An update statement on R will produce history data and save in same table. Allow queries by default to be on current data.

P15. Implement simple moving object DB, in which a relation may contain column for speed (x and y direction) of an object like a ship and time duration in which this speed is maintained. An object will have multiple such tuples representing its speed in the past. Implement:
- functions to get position of X at time T
- function to find items which will be very close (within distance, say, 5) in next 30 min