20

An Architectural Framework

Michael D. Soo, Christian S. Jensen, and Richard T. Snodgrass

1 Introduction

In this chapter we address the question of how to construct a DBMS architecture supporting TSQL2. Our goal is to enumerate the changes that a conventional DBMS would need to support TSQL2. We are concerned with modifying a conventional DBMS in a minimal fashion to support TSQL2. While a more elaborate architecture is possible (likely with significant performance gains), our purpose is to describe the "first step" towards the realization of a temporal DBMS.

In the next section, we describe a canonical design for a conventional DBMS. We then describe the minimal changes needed by each component of the architecture to support TSQL2. We conclude with a few observations about the described architecture.

2 Conventional Architecture

Figure 1 shows a conventional DBMS architecture supporting SQL-92. In the figure, ovals represent data items, e.g., user submitted queries, boxes represent software components, e.g., the query compiler, and arrows show the flow of data through the DBMS.

Queries may be submitted by four types of users, the database administration (DBA) staff, interactive users, application programmers, and parametric users.

The DBA staff is responsible for defining and maintaining the database through the execution of data definition language (DDL) statements and privileged commands not available to other users.

Interactive users are sophisticated, database-literate users. They submit SQL-92 queries which are compiled by the query compiler into an procedure-oriented internal representation, the query execution plan. The query execution plan is passed to the run-time evaluator for execution. Actual access to the stored data is performed by the transaction and data manager.

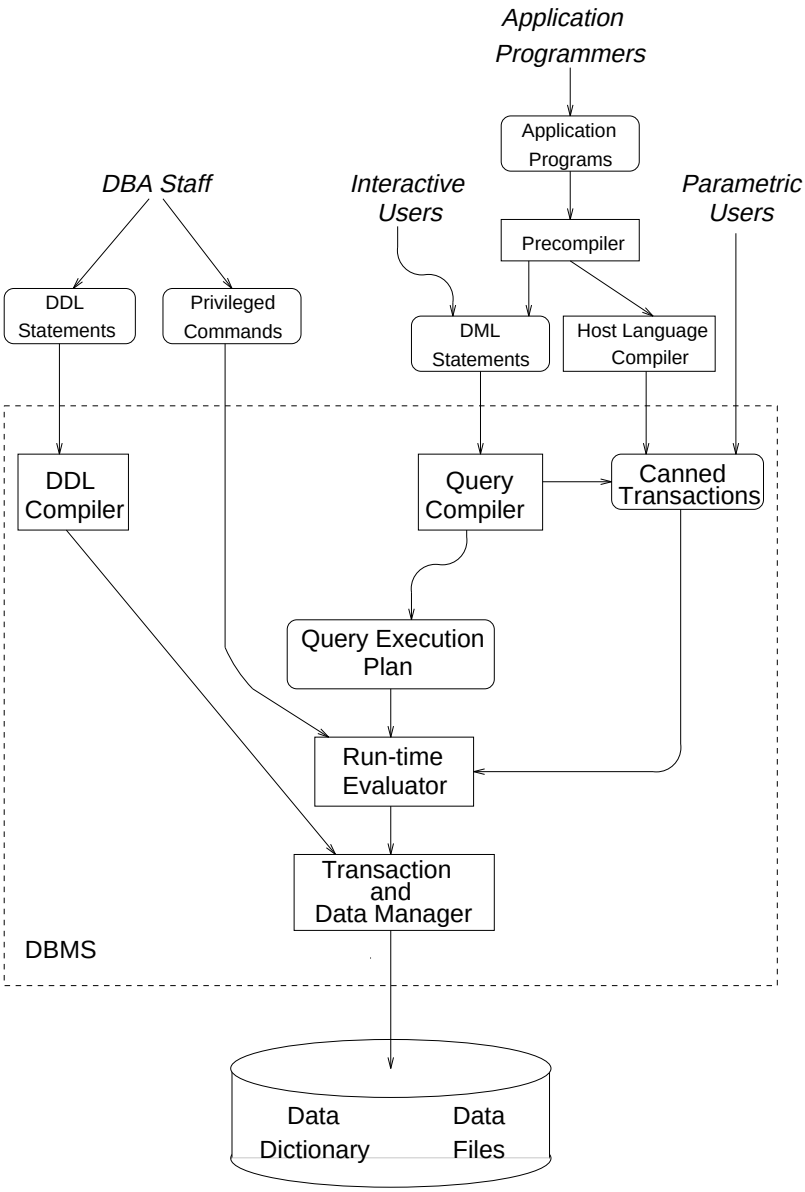

Figure 1: Canonical DBMS architecture

Application programmers submit application programs written in a host programming language, e.g., C, that contain embedded database queries. A precompiler separates the embedded queries from the host application and routes each to an appropriate compiler. The compiled database query and host program are recombined to produce a canned transaction, which may be executed at some later date.

Parametric users are unsophisticated users such as airline reservation agents or customer service representatives. They use the canned transactions produced by applications programmers.

Example 1 Consider snapshot versions of the Employee and Manages schemas defined in Chapter 21, Section 4.1. We illustrate the flow of information through the architecture with the following query, which returns the names of all employees managed by Bob.

```
SELECT Employee.Name, Manages.MgrName
FROM Employee, Manages
WHERE Manages.Dept = Employee.Dept AND
      Manages.MgrName = 'Bob'
```

Suppose that this query is submitted by an interactive user. (We could assume that the query was submitted via another path, e.g., as an embedded query in an applications program, but the discussion would be more complicated, and no more illustrative.) The query, a DML statement, is first processed by the *query compiler*. The query compiler analyzes the query, first syntactically and then semantically. Syntactic analysis ensures that the lexical and syntactic structure of the query is correct. Semantic analysis performs type checking and verifies that other semantic constraints are satisfied. Though not shown on the diagram, schema information contained in the data dictionary is used during semantic analysis.

Ultimately, the query compiler produces a procedural expression of the submitted query that is suitable for execution by the run-time evaluator. This procedural expression is based on the relational algebra. As an intermediate step, the query compiler translates the query into a simple algebraic expression which is then optimized. Such an expression for our example might be the following.

$$\pi_{\{Employee.Name, Manages.MgrName\}} (Employee \bowtie_{Employee.Dept=Manages.Dept} (\sigma_{MgrName='Bob'}(Manages)))$$

The final result produced by the query compiler is a query execution plan, which is essentially the optimized algebraic expression with specific algorithms chosen for each algebraic operation. For example, the query compiler might generate the following query execution plan, depending on a estimate of the cost of various algorithms implementing the different operators.

$$temp_1 \leftarrow index\_select(Manages, 'Bob')$$
$$result \leftarrow project_{\{Name, MgrName\}} (nested\_loop\_join(Employee, temp_1, 'Dept'))$$

In this query execution plan, the query compiler makes use of an existing index on the MgrName attribute of the Manages table to quickly find the departments that Bob manages. This intermediate result is stored in the temporary table $temp_1$. As $temp_1$ is likely to fit in main memory, i.e., Bob only manages a few departments, and hence $temp_1$ will contain only a few tuples, a simple nested-loop join is used to find Bob's employees. In addition, rather than writing another temporary result,

the compiler chooses to perform the final projection of the employee name and the manager name attributes "on the fly" with the join.

The query execution plan is sent to the run-time evaluator for execution. For example, for the *index_select* operation in the above query execution plan, the run-time evaluator executes this algorithm, generating a series of index retrievals and subsequent table page retrievals to materialize the selection. The data retrieval operations are performed by the transaction and data manager which manages the buffer space alloted to the transaction, and ensures the consistency of the database even though multiple transactions may be executing concurrently in the DBMS. □

In the remainder of this section, we describe the minimal changes required to each DBMS component in order to support TSQL2. We note that the precompiler and host language compiler are largely independent of the database query language—they require only small changes to support temporal literal/timestamp conversion. For each of the remaining components, the data dictionary and data files, as well as those within the DBMS proper, we describe the minimal modifications needed by these components to support TSQL2 queries.

3 Data Dictionary and Data Files

The data dictionary and data files contain the database, the actual data managed by the DBMS. The data dictionary records schema information such as file structure and format, the number and types of attributes in a table, integrity constraints, and associated indexes. The data files contain the physical tables and access paths of the database.

For a minimal extension, the data files require no revision. We can store tuple-timestamped temporal tables in conventional tables, where the timestamp attributes are stored as explicit atomic attributes. However, the data dictionary must be extended in a number of ways to support TSQL2 [1]. The most significant extensions involve schema versioning, multiple granularities, and vacuuming.

For schema versioning, the data dictionary must record, for each table, all of its schemas and when they were current. The data files associated with a schema must also be preserved. This is easily accomplished by making a transaction-time table recording the schemas for a single table. The transaction time associated with a tuple in this table indicates the time when the schema was current.

Multiple granularities are associated in a lattice structure specified at system generation time. A simple option is to store the lattice as a data structure in the data dictionary. Alternatively, if the lattice is fixed, i.e., new granularities will not be added after the DBMS is generated, then the lattice can exist as a separate data structure outside of the data dictionary.

Vacuuming specifies what information should be physically deleted from the database. Minimally, this requires a timestamp, the cut-off time, to be stored for each transaction-time or bitemporal table cataloged by the data dictionary. The cut-off time indicates that all data current in the table before the value of the timestamp has been physically deleted from the table.

4 DDL Compiler

The DDL compiler translates TSQL2 `CREATE`, `ADD`, `REPLACE` and `DROP` statements [1] into executable transactions. Each of these statements affects both the data dictionary and the data files. The `CREATE` statement adds new definitions, of either tables or indexes, to the data dictionary and creates the data files containing the new table or index. The `ADD` and `REPLACE` statements change an existing schema by updating the data dictionary, and possibly updating the data file containing the table. The `ADD` statement is used to add new columns or indexes to a schema, and the `REPLACE` statement is used to change an existing column or index. Lastly, the `DROP` statement is used to remove a table, column, or index definition from a schema, as well as the actual physical data.

Numerous changes are needed by the DDL compiler, but each is straightforward and extend existing functionality in small ways. First, the syntactic analyzer must be extended to accommodate the extended TSQL2 syntax for each of the `CREATE`, `ADD`. `REPLACE`, and `DROP` statements. The semantic analyzer must be extended in a similar manner, e.g., to ensure that an existing table being transformed into a valid-time state table via the `ADD VALID STATE` command is not already a valid-time table.

5 Query Compiler

The query compiler translates TSQL2 data manipulation language (DML) statements into an executable, and semantically equivalent, internal form called the query execution plan. As with the DDL compiler, each phase of the query compiler, syntactic analysis, semantic analysis, and query plan generation, must be extended to accommodate TSQL2 queries.

We use the model that the initial phase of the compilation, syntactic analysis, creates a tree-structured query representation which is then referenced and augmented by subsequent phases. Abstractly, the query compiler performs the following steps.

1. Parse the TSQL2 query. The syntactic analyzer, extended to parse the TSQL2 constructs, produces an internal representation of the query, the parse tree.

2. Semantically analyze the constructed parse tree. The parse tree produced by the syntactic analyzer is checked for types and other semantic constraints, and simultaneously augmented with semantic information.
3. Lastly, a query execution plan, essentially an algebraic expression that is semantically equivalent to the original query, is produced from the augmented parse tree by the query plan generator.

The minimal changes required by the query compiler are summarized as follows.

- The syntactic and semantic analyzers must be extended to support TSQL2.
- The query execution plan generator must be extended to support the extended TSQL2 algebra, including the new coalescing, join, and slicing operations. In a minimally extended system, it may be acceptable to use existing algebraic equivalences for optimization, even with the extended operator set. Such an approach preserves the performance of conventional snapshot queries. Later inclusion of optimization rules for the new operators would be beneficial to the performance of temporal queries.
- Support for vacuuming must be included in the compiler. Query modification, which normally occurs after semantic analysis and prior to query optimization, must be extended to include vacuuming support.

The need to extend the syntactic and semantic analyzers is self-evident, and straightforward. (A query compiler has been implemented in conjunction with the MULTICAL project that syntactically and semantically analyzes a significant subset of TSQL2.) Extending the query plan generator to use the extended algebra is also straightforward, assuming that temporal aspects of the query are not considered during query optimization. In the worst case, the same performance would be encountered when executing a temporal query on a purely snapshot database. Lastly, in order to support vacuuming, the query compiler, within its semantic analysis phase, must support automated query modification based on vacuuming cut-off times stored in the data dictionary.

6 Run-time Evaluator

The run-time evaluator interprets a query plan produced by the query compiler. The run-time evaluator calls the transaction and data manager to retrieve data from the system catalog and data files.

We assume that the run-time evaluator makes no changes to the query plan as received from the query compiler, i.e., the query plan, as generated by the query compiler, is optimized and represents the best possible evaluation plan for the query. As such the changes required for the run-time evaluator are small. Particularly, since evaluation plans for the any new operators have already been selected by the query

compiler, the run-time evaluator must merely invoke these operations in the same manner as non-temporal operations. Additionally, as will be mentioned in Chapter 21, Section 5, evaluation algorithms for the new temporal operators (coalescing, n-way joins, and slicing) are similar to well-known algorithms for snapshot operators. For example, coalescing can be implemented with slightly modified duplicate elimination algorithms, which have been well-studied in snapshot databases.

Lastly, changes are needed by the run-time evaluator to support the input and output of temporal literals, as discussed in [2, Chapter 8]. Calls to the software components supporting temporal literals must be inserted into the query execution plan by the query compiler and subsequently performed by the run-time evaluator.

7 Transaction and Data Manager

The transaction and data manager performs two basic tasks: it manages the transfer of information to and from disk and main memory, and it ensures the consistency of the database in light of concurrent access and transaction failure.

Again, at a minimum little needs to be modified. We assume that the conventional buffer management techniques are employed. Supporting transaction time requires the following small extension to the concurrency control mechanism.

For correctness, transaction times are assigned at commit time, otherwise during an interleaved execution a transaction may see data that is not yet current. This would happen if a transaction reads tuples previously written by a concurrent transaction holding a later transaction time. To avoid this problem, we have an executing transaction write tuples without filling in the transaction timestamp of the tuples. When the transaction later commits, the transaction times of affected tuples are then updated.

This is accomplished by maintaining a (reconstructible) table of tuple IDs written by the transaction. This table is read by an asynchronous background process which performs the physical update of the tuples' transaction timestamp. Correctness only requires that the transaction times for all written tuples be filled in before they are read by a subsequent transaction. While this simple extension suffices, more complex and efficient methods have been proposed [3]. Notice also that this algorithm does not affect the recovery mechanism used by the DBMS, assuming that the transaction time of a committed transaction is logged along with the necessary undo/redo information.

8 Summary

We have described how a canonical DBMS architecture can be extended to support TSQL2. The changes described are minimal in that they represent the smallest

necessary extensions to support the functionality of TSQL2. As the extensions are small, we believe that, as a first-step, TSQL2 can be supported for a relatively low development cost.

We anticipate that the performance of the minimally extended architecture will rival the performance of conventional systems. Snapshot queries on the current database state may suffer a slight performance penalty due to the additional temporal support. However, since we are able to use existing optimization techniques, evaluation algorithms, and storage structures, we expect snapshot queries on the temporal DBMS to approach the performance of identical queries on a conventional DBMS.

Conversely, while there are many opportunities for improvement, we believe that temporal queries on the minimally extended architecture will show reasonable performance. In particular, the architecture can employ new evaluation and optimization techniques for temporal queries currently under investigation. With the addition of temporally optimized storage structures, we expect further performance improvements.

References

[1] Snodgrass, R. T., I. Ahn, G. Ariav, D.S. Batory, J. Clifford, C.E. Dyreson, R. Elmasri, F. Grandi, C.S. Jensen, W. Käfer, N. Kline, K. Kulkarni, T.Y.C. Leung, N. Lorentzos, J.F. Roddick, A. Segev, M.D. Soo and S.M. Sripada. "TSQL2 Language Specification." *ACM SIGMOD Record*, 23, No. 1, Mar. 1994, pp. 65–86.

[2] Snodgrass, R. T. (editor) "The TSQL2 Temporal Query Language." Kluwer Academic Publishers, 1995.

[3] Lomet, D. and B. Salzberg. "Transaction-Time Databases," in Temporal Databases: Theory, Design, and Implementation. Benjamin/Cummings, 1993. Chap. 16. pp. 388–417.