The TSQL Benchmark
DRAFT

James Clifford Shashi K. Gadia Fabio Grandi Christian S. Jensen
Patrick Kalua Sunil Nair Edward Robertson John F. Roddick
Maria Rita Scalas Richard T. Snodgrass Abdullah Tansel Paolo Tiberio
Alexander Tuzhilin Gene Wuu

May 8, 1993

1 Introduction

1.1 Goal

The central goal of this document is to provide the temporal database community with a *comprehensive consensus benchmark* for temporal query languages that is *independent* of any existing language proposal.

This is not a performance benchmark, but is rather a *semantic* benchmark intended to be an aid in evaluating the user-friendliness of proposals for temporal query languages. Thus, temporal query languages should ideally be able to express the benchmark queries both conveniently and naturally.

To obtain a consensus benchmark, researchers in temporal databases have been invited to participate in this initiative, and each researcher that has contributed significantly will be a coauthor. The electronic mail distribution `tsql@cs.arizona.edu` is used as the medium for discussing the benchmark and related issues.

As a consequence of the central goal above, no existing temporal data models are used or mentioned. The relation schemas of the benchmark are expressed as sets of attributes, including one attribute illustrating user-defined time. However, the underlying temporal aspects are implicit (of course, specific temporal data models might add explicit temporal attributes). The contents of the relations are described in natural language. The benchmark queries are also given only in natural language.

The benchmark is *not* intended to constitute a metric for query language completeness, and as such it is not a substitute for a rigorous *theoretical* study of expressive powers of various temporal query languages. Comprehensiveness of the benchmark is desirable only to ensure that all aspects of query language design are covered.

It it emphasized that using the benchmark as an advanced, quantitative scoring system for comparing languages makes little sense. Thus, one language is not necessarily superior to another just because one is capable of expressing more benchmark queries than the other. Rather, the focus is on user-friendliness.

1.2 Scope

Within certain boundaries, discussed next, the benchmark is intended to contain all queries that appear reasonable and that are consistent with the schema and data of the benchmark.

First, the benchmark is of a semantic nature—in its current form, it is not aimed at performance comparisons. The intention is to provide a foundation for comparing the descriptive and operational characteristics and capabilities of temporal query languages, not their performance characteristics. Properly extended with additional relation schemas and a variety of large instances, the benchmark can also be used for performance comparisons.

Second, a number of restrictions are imposed on which types of queries are admissible in this version of the benchmark, including the following.

- Queries are restricted to valid time only. Transaction-time related queries are not explored.
- Schema evolution and versioning are not considered.
- Incompleteness is not considered.
- Recursive queries are not included.
- General temporal reasoning is beyond the scope of this version of the benchmark.
- Queries involving aggregation facilities are not considered.
- Only queries are included—updates are not considered.
- Continuous attributes such as time are not included.
- The querying of data valid in the future is not explored.

These advanced aspects are excluded solely for pragmatic reasons, and the exclusion is not meant to imply in any way that the aspects are not important. The restrictions simply represent an attempt to reduce the size of the initial benchmark to manageable proportions.

It is emphasized that this benchmark is merely the first in a sequence of ever-more comprehensive benchmarks. Later benchmarks will relax the above restrictions on the scope of comprehensiveness imposed on this benchmark. Specifically, the next version of the benchmark is intended to include queries that involve aggregation.

2 The Benchmark Database Schema

2.1 Criteria

A suitable database schema for the benchmark should satisfy four criteria.

- The schema should be natural. That is, it should correspond to a reasonable, though possibly greatly simplified, segment of the real world. This both reduces the need to explain the model and enhances the ability to recognize verball pitfalls in the path to the query instances.
- The schema should be simple. This will aid in making the benchmark easy to understand. This criterion restricts the number of relation schemas and the number of attributes of the individual schemas. Additionally, the names of the relations and of the attributes should be short, as they will be referenced repeatedly.

  When an extension is proposed, the benefits should be carefully compared with the added complexity.

- The schema should allow for comprehensiveness within the chosen scope. Using the schema, it should be possible formulate queries of all the types that appear reasonable.

  This indicates a need for at least two related relation schemas (for natural join queries).

- A schema that has already been used frequently is preferred over a new schema. This guarantees that many existing queries can be adapted easily to the benchmark.

- For clarity, schema and attribute names must start with capital letters.

2.2 The Schema

The database schema consists of three valid-time relation schemas, `Emp`, `Skills`, and `Dept`. They are defined as follows.

Relation `Emp` uses the attributes `Name`, `Salary`, and `Dept` for recording the salaries of employees and the departments where they work. In addition, it contains attributes `Gender` and `D-birth` which indicate the gender and date of birth of an employee. While the salary and department of an employee varies over time, both `Gender` and `D-birth` are time-invariant.

Relation `Skills` records the association of employees with skills via the two attributes `Name` and `Skill`. The skills of an employee may vary over time. For example, employees are considered to have the skill "driving" only during those interval(s) when they hold valid licenses.

Relation `Dept` records the association of employees, as managers, with departments, and it contains three attributes, `Department`, recording a department name, `Manager`, recording the manager of the department, and `Budget`, recording the budget of the department.

Attributes `Name`, `Dept`, `Department`, `Skill`, and `Manager` are of type `textString`; attribute `Gender` is one of `F` (female) and `M` (male); `Salary` and `Budget` are of type `integer`; and `D-birth` is a user-defined time value which may be compared with valid times.

The relation schemas obey the following *snapshot* functional and multivalued dependencies:

For `Emp`:
$\texttt{Name} \rightarrow \texttt{Salary}$
$\texttt{Name} \rightarrow \texttt{Dept}$
$\texttt{Name} \rightarrow \texttt{Gender}$
$\texttt{Name} \rightarrow \texttt{D-birth}$
For `Skills`:
$\texttt{Name} \twoheadrightarrow \texttt{Skill}$ (and $\texttt{Name} \not\rightarrow \texttt{Skills}$)
For `Dept`:
$\texttt{Department} \rightarrow \texttt{Manager}$
$\texttt{Manager} \rightarrow \texttt{Department}$
$\texttt{Department} \rightarrow \texttt{Budget}$

Note that `Name` is the primary key of `Emp` (it is the only candidate key). For `Skills`, there is no non-trivial key. For `Dept`, each of `Department` and `Manager` is a candidate key, and `Department` is selected as the primark key.

Each of the relation schemas are in snapshot Boyce-Codd normal form.

It is emphasized that the notion of key does not capture correspondence between attribute values and the real-world objects they represent. As one consequence, it is possible in this schema, e.g., for a person to change `Name` attribute value over time.

The attribute `Manager` of `Dept` is a foreign key for the attribute `Name` of `Emp`. Thus, a tuple is allowed to exist in the `Dept` relation only if, for each non-empty snapshots of this tuple, the `Manager` attribute value exists as a `Name` value of some tuple in the simultaneous snapshot of the `Emp` relation.

3 The Benchmark Data

3.1 Criteria

- For clarity, the database instance should ideally accord with *all and only* those constraints which are explicitly stated in the definition of the database schema.
- For simplicity and ease of typing, attribute values should be short and salary values should be multiples of $10,000.
- Transitions (i.e., timestamp values) occur only at the beginning of the month, and all dates should be in the time interval from 1/1/81 to 12/31/88 (because the digits 8 and 9 are relatively hard to distinguish). Time intervals are all specified by the inclusive starting and ending events. Also for clarity, relation instance names should start with lowercase letters.
- The data should include a "hole in the history" of some entity. For example, the database may be designed to contain a whole in the employment of some employee.
- The data should include asynchronous behavior of attributes. For example, the department and salary of employees may change independently.

3.2 The Proposed Data

Three instances, `emp`, `skills`, and `dept`, are defined over the `Emp`, `Skills`, and `Dept` relation schemas, respectively. The contents of these instances is described below.

There are two employees, identified by *ED* and *DI* in the following.

ED worked in the Toy department from 2/1/82 to 1/31/87, and in the Book department from 4/1/87 to the present. His name was Ed from 2/1/82 to 12/31/87, and Edward from 1/1/88 to the present. His salary was $20K from 2/1/82 to 5/31/82, then $30K from 6/1/82 to 1/31/85, then $40K from 2/1/85 to 1/31/87 and 4/1/87 to the present. *ED* is male and was born on 7/1/55. Several skills are recorded for *ED*. He has been qualified for typing since 4/1/82 and qualified for filing since 1/1/85. He was qualified for driving from 1/1/82 to 5/1/82 and from 6/1/84 to 5/31/88.

DI worked in and managed the Toy department from 1/1/82 to the present. Her name is Di throughout her employment. The budget of the Toy department was $150K from 1/1/82 to 7/31/84, $200K from 8/1/84 to 12/31/86, and $100K from 1/1/87 to the present. Her salary was $30K from 1/1/82 to 7/31/84, $40K from 8/1/84 to 8/31/86, then $50K from 9/1/86 to the present. *DI* is female and was born on 10/1/60. *DI* has been qualified for directing from 1/1/82 to the present.

The present time (i.e., the value of `now`) is 1/1/90.

4 Classification of Benchmark Queries

A classification of benchmark queries will be based on a comprehensive taxonomy of queries. First, critria for such a taxonomy are outlined. Next, the taxonomy itself is presented. As the taxonomy is too fine-grained, categories are then merged into an adequate number of groups which can subsequently be used for classification.

4.1 Criteria

Three criteria for an appropriate taxonomy of benchmark queries are suggested.

- The taxonomy should be schema and instance independent. This criterion helps ensure that the taxonomy will persist when the benchmark database schema evolves as new versions appear. Ideally, this will allow for an incremental mode of work, where only new queries need to be categorized and existing queries do not need re-categorization.
- The taxonomy should provide comprehensive coverage of benchmark queries. Comprehensiveness is desirable to avoid holes and point to many categories of queries.
- The taxonomy should be useful when structuring the presentation of benchmark queries. Most importantly, it should provide sufficient structure. Thus, taxonomies that have only few categories and that map many queries to single categories are problematic. If the number of categories is excessive for presentation purposes, classes of categories may be identified with individual sections.

4.2 The Taxonomy

The taxonomy is characterized as having a projection (output) and a selection component, much like SQL. Then each component is covered in turn. Finally, the full taxonomy is summarized and a notation for naming individual categories is defined.

4.2.1 Top-level Taxonomy

At the top level, the taxonomy is divided into two orthogonal parts, namely a part where queries are categorized according to their *output component* and a part where the categorization is based on the *selection component*. Thus, a category is described by two components, as illustrated in Figure 1.

$$\{< \text{output component} >\} \times \{< \text{selection component} >\}$$

Figure 1: Top-level Description of Benchmark Taxonomy

This top-level design reflects the SQL template (i.e., `SELECT` ... `FROM` ... `WHERE` ...). The first component categorizes the contents of the `SELECT` clause, and the second component categorizes the contents of the `WHERE` clause. No component is needed to reflect the `FROM` clause where tuple variables are defined. The two components are orthogonal only in the same sense that the `WHERE` and `SELECT` clauses of a particular query are orthogonal.

4.2.2 Output-based Taxonomy

The output-based taxonomy is intended to reflect the part of queries where the format of the resulting tuples is specified. The taxonomy is described in Figure 2 and is explained in the following.

The idea is to distinguish between queries based on the format of the result tuples. A tuple may include an explicit-attribute component and a valid-time component, each of which are considered next.

$$\left\{\begin{array}{c}\underline{\text{explicit-attribute component}}\\ \text{none}\\ \text{projected}\\ \text{complete}\end{array}\right\} \times \left\{\begin{array}{c}\underline{\text{valid-time component}}\\ \text{none}\\ \left\{\begin{array}{c}\underline{\text{type}}\\ \text{event}\\ \text{interval}\\ \text{element}\end{array}\right\} \times \left\{\begin{array}{c}\underline{\text{value}}\\ \text{derived}\\ \text{imposed}\end{array}\right\}\end{array}\right\}$$

Figure 2: Output-based Taxonomy

If present, the explicit-attribute component, may contain all attributes in the argument relation (multiple relations are discussed below) or it may contain a subset of the attributes in the argument relation. In the first case, the explicit attribute component is "complete," and in the second, it is "projected."

To exemplify, consider a tuple telling that Ed is in the Book department from 1/1/82 to 12/31/84. Here "Ed" and "Book" constitute the explicit-attribute component, and "1/1/82" and "12/31/84" is the valid-time component. If the argument relation contained an attribute "Salary" in addition to the Name and Department attributes, this result is projected.

If several relations are used in a query, the argument relation is the Cartesian product of these, i.e., the schema is the concatenation of the schemas of the relations used in the query.

The valid-time component of a tuple may be of three types. First, it may be an event, i.e., a single time value (e.g., 3/1/83). Second, it may be an interval, i.e., a sequence of consecutive time values (e.g., as above). Third, it may be an element, i.e., a set of time values which may be described by a set of intervals (e.g., 1/1/82 to 12/31/84, 2/1/85 to 3/31/85, and 5/1/86 to 5/31/86).

Orthogonally, the value of a valid-time component may be derived or imposed. A derived value is computed solely in terms of the valid-time components of the tuples in the argument relation. An imposed value is computed by explicit assignment in the query.

Note that at least one of the two components must be present in the result of a query. This part of the taxonomy results in 20 mutually exclusive categories.

The distinctions above are based on the schema of result relations. It is possible also to distinguish between the cardinalities of result relations, e.g., between set-valued and single-tuple valued results.

4.2.3 Selection-based Taxonomy

The selection component is divided into two parts, one for valid-time selection and one for selection not involving valid time. See Figure 3.

$$\{<\text{valid-time selection}>\}^* \times \{<\text{non-temporal selection}>\}^*$$

Figure 3: Top-level Selection-based Taxonomy

Both parts are based on the same observation. In general, a selection predicate is built from atomic selection predicates using logical operators (e.g., `and`, `or`, and `implies`) and parenthesis.

Both parts categorize queries based on the atomic predicates used in the queries. As several types of atomic predicates may be used in the same query, queries generally fall into multiple categories (as indicated in Figure 3 by the Kleene star, "*"). We examine each part of the selection-based taxonomy in turn.

Atomic valid-time selection predicates are assumed to be of the form

$$arg_1 \ \texttt{op} \ arg_2 \ ,$$

where `op` is a some comparison operator (e.g., `precedes`, and `contains`). It is assumed that arg_1 is the valid time of the data, and restrictions are imposed based on the type of the comparison operator, on the origin of arg_2, and on the type of arg_2. Figure 4 outlines the categories.

$$\left\{ \begin{array}{c} \underline{\text{type of comparison operator}} \\ \text{duration-based} \\ \text{ordering-based} \\ \text{containment-based} \end{array} \right\} \times \left\{ \begin{array}{c} \underline{\text{type of } arg_2} \\ \text{event} \\ \text{interval} \\ \text{element} \end{array} \right\} \times \left\{ \begin{array}{c} \underline{\text{origin of } arg_2} \\ \text{explicitly supplied in query} \\ \text{user-defined attribute value} \\ \text{computed from other valid times} \end{array} \right\}$$

Figure 4: Valid-time Selection-based Taxonomy

Three types of comparison operators are identified. First, a comparison operator may be duration-based. For example the operator `spanExceeds` returns true if the duration of the first argument is equal to or larger than the duration of the second argument. Second, comparison operators may be based on ordering. Operators in this category include `precedes` and `meets`. The first applies to all timestamps and evalutes to true if the largest time in the first argument is smaller than the smallest times in the second argument. Operator `meets` appears to be useful only for events and intervals. Two timestamps meet if they are not separated by any event (i.e., may be coalesced). Operators based on containment include `=` (identity), `overlaps`, and `contains`.

The second argument (arg_2) may be an event, an interval, or an element. Also, it may come from three sources. First, it may be supplied directly in the query, as a constant. Second, it may be the value of a user-defined time attribute in an argument tuple. Note that this is only possible for events if first normal form is required. Third, like the first argument, the second argument may be computed from valid times in the argument tuples.

If the three types of categories are completely orthogonal, this part of the taxonomy will contribute with a total of 27 categories. However, it may be debated whether intervals and elements may be used as values of user-defined attributes (resulting in non-1NF relations).

The final part of the selection-based taxonomy categorizes queries based solely on the part of the selection component that involves only ordinary, non-temporal selection.

Many possibilities for categorization exist. Below, in Figure 5, we distinguish between four significant types of atomic selection predicates. First, an attribute may be compared with a constant, supplied by the user. Second, attribute values, both in the same relation, may be compared. Third, a primary key value may be compared with a matching foreign key value. Fourth, arbitrary attributes of possibly distinct relations may be compared. In the figure, $\theta ::= < \mid > \mid \leq \mid \geq \mid =$, i.e., a combination of equality and/or the one of the two inequality operators. If we distinguish between situations where only equality is involved and situations where inequality is involved, this give 8 categories.

$$\left\{ \begin{array}{c} \underline{\text{non-temporal attribute value selection}} \\ att\ \theta\ Constant \\ att_1\ \theta\ att_2 \\ att_k\ \theta\ att_{fk} \\ att(rel_1)\ \theta\ att(rel_2) \end{array} \right\} \times \left\{ \begin{array}{c} \underline{\text{comparison operator, } \theta} \\ \text{only equality } (=) \\ \text{inequality } (<>) \end{array} \right\}$$

Figure 5: Non-temporal Selection-based Taxonomy

4.2.4 Additional Contributions—TEMPORARY

The distinction between grouped and ungrouped queries has not been integrated into the taxonomy. To do that, definitions of these categories are needed.

4.3 Overview and Naming of Categories

Each query has a single output component, zero or more valid-time selection components (one per such operator), and zero or more non-temporal selection-based components (one per such operator). The taxonomy is summarized in Figure 6. There, the names introduced in the taxonomy are used along with punctuation in order to name a category.

```
<category>        ::= <output> '/' {<v-t selection> }* '/' {<non-t selection> }*
<output>          ::= '(' {None | Projected | Complete } ','      /* explicit-attribute component
                      {None |                                     /* no valid-time attribute
                       {Event | Interval | Element } ','          /* type of valid-time attribute
                       {Derived | Imposed } } ')'                 /* value of valid-time attribute
<v-t selection>   ::= '(' {Duration | Ordering | Containment } ','   /* operator type
                      {Event | Interval | Element } ','              /* argument type
                      {Explicit | User-defined | Computed } ')'      /* argument origin
<non-t selection> ::= '(' {'=' | '<>' } ','                          /* operator type
                      {Constant | Single | Foreign | Arbitrary } ')' /* argument types
```

Figure 6: Overview of the Taxonomy used for Naming Categories

To exemplify the use of Figure 6 for naming categories, consider the query "When was Ed Manager of the Toy Department." This query is in the category shown next (with no valid-time selection).

(None, Element, Derived) // (=, Constant)

It may be observed that the taxonomy gives rise to a large number of categories. For example, assuming a single non-temporal operator and no valid-time operators, there are $20 \times 8 = 160$ categories. Adding a single valid-time operator while assuming orthogonality yields an additional 4320 categories!

As a result, it becomes necessary to create classes of categories which then may be used for clasifying the benchmark queries.

One approach would be to name a *class* of categories of queries, by simply replacing one or more of the entries with the Kleene star ("*"), e.g.,

(None, Element, Derived) / (*,*,*) / (=, Constant)

The above query category would be in this class. In the next section, we define the classes to be used in the benchmark.

4.4 Forming Classes from Categories

The idea is to remove distinctions from the comprehensive taxonomy until a suitable number of classes is obtained. Figure 7 is thus a reduced version of Figure 6.

<class-name>	::= <reduced output> '/' {<reduced v-t selection> }*	
<reduced output>	::= '(' {None | Proj/Comp } ','	/* explicit-attribute component
	{None | Not empty } ')' '/'	/* valid-time attribute component
<reduced v-t selection>	::= '(' {Duration | Other } ','	/* comparison operator type
	{Event | Interval | Element } ','	/* argument type
	{Computed | Other } ')'	/* argument origin

Figure 7: Overview of the Classification of Queries

The second and third lines concern output. Only the prescence or absence of explicit attributes and timestamps are distinguished, leading to three categories. The last three lines concern valid-time selection (non-temporal selection is disregarded). Comparison operators may be duration-based or not; arguments be of either event, interval, or element type; and the arguments may or may not derive from valid times of tuples.