INSTITUTE FOR ELECTRONIC SYSTEMS
AALBORG UNIVERSITY

DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE

FREDRIK BAJERS VEJ 7 ■ DK 9220 AALBORG Ø ■ DENMARK

Phone + 45 98 15 85 22
Telex 69790 aub dk
Telefax + 45 98 15 81 29

Tiger

Reference Manual[1]

Michael Böhlen

boehlen@cs.auc.dk

Tiger is an period-timestamped bitemporal database system running as a frontend to the commercial (relational) database system Oracle. It implements ATSQL [BJ96], a temporal extension of SQL. Temporal requests (i.e., ATSQL commands) are compiled into SQL-commands that are executed by the commercial database backend.

Special care was given to the design of a system that allows for a seamless integration of time into legacy database application. Specifically, this means that Tiger allows you to use existing applications without a single change. This was achieved by making upward compatibility and temporal upward compatibility design goals. Next, the concept of sequentiality ensures a wholesale and powerful support for temporal database requests based on snapshot reducibility. Finally the concept of nonsequential statements permits the formulation of arbitrary temporal relationships.

Tiger fully exploits state-of-the-art technology to make its services available to an international community. HTML, CGI, JavaScript, and Expect are used to support interactive accesses through the world wide web.

[1] Draft version. Not complete yet.

Contents

1 Introduction **3**
1.1 Copyright and Warranty 3

2 Access & Initialization **4**

3 System Architecture **6**

4 ATSQL **9**
4.1 Data Model 9
4.2 Crucial concepts 10
4.2.1 Upward Compatibility (UC) 10
4.2.2 Temporal Upward Compatibility (TUC) 10
4.2.3 Sequentiality (SEQ) 11
4.2.4 Nonsequentiality (NONSEQ) 13
4.3 Syntax 13
4.3.1 Reserved Words 13
4.3.2 Temporal Built-Ins 14
4.3.3 Flags 15
4.4 Semantics 19

5 Oracle MetaDB Tables **26**

6 Command Overview **28**
6.1 Miscellaneous Commands 28
6.2 ATSQL Commands 29

7 Test Files **36**
7.1 Migrating Databases 36
7.2 Data Manipulation Statements 39

8 Conclusions and Future Plans **41**

9 Acknowledgements **42**

Bibliography **42**

Index **43**

Chapter 1

Introduction

Tiger has been designed and developed to promote and further the understanding of temporal databases in general and ATSQL, a temporal extension of SQL, in particular. The main goals were to exploit existing database technology and to achieve substantial support for temporal requests with little syntactic overhead.

1.1 Copyright and Warranty

Chapter 2

Access & Initialization

In contrast to other software packages downloadable over the internet there's no installation required to use Tiger. It is freely accessible through the world wide web and the necessary initialization steps have been performed before-hand. Connect to either `http://www.cs.auc.dk/~boehlen` or `http://www.cs.auc.dk/general/DBS/software`. There you find links to the Tiger system. You're of course also welcome to add a reference to one of your own web pages. The recommended way is to add some JavaScript code to your home page. This is very simple. First, add

```
<SCRIPT LANGUAGE="JavaScript">
<!--
  function start_tiger() {
    tiger = window.open(
       'http://www.cs.auc.dk/~boehlen/Software/Tiger/x.html',
       'tiger',
       'toolbar=no,location=no,directories=no,status=yes,\
        menubar=no,scrollbars=yes,resizable=yes,\
        copyhistory=no,width=700,height=700');
  }
//-->
</SCRIPT>
```

to your HTML header and then add a reference of the form

```
<A HREF="" OnClick="start_tiger()">Tiger</A>
```

to the body of your HTML document. This starts up Tiger in a separate, frill-less window. Of course it is also possible to reference Tiger directly via the URL `http://www.cs.auc.dk/~boehlen/Software/Tiger/x.html`.

For Tiger to run a few meta tables are required. When you connect, a database account with these tables predefined is automatically assigned to you. Although there are basic checks to prevent these tables from being manipulated or dropped it is still possible to do so. (Tiger addresses the research community and functionality is given priority over security.) If this is done by someone and you happen to get assigned the "corrupted" account later on you might not find these tables present. In this case please issue the following statements to create the meta tables.

```
CREATE TABLE TDB$ICS (
  NAME VARCHAR2(32) PRIMARY KEY,
  QUERY LONG
);

CREATE TABLE TDB$DEPS (
  OBJECT VARCHAR(32),
  TYPE VARCHAR(32),
  BASE_OBJECT VARCHAR(32),
  IS_VIEW CHAR(1),
  IS_PROHIBITIVE CHAR(1)
);

CREATE SEQUENCE TDB$S0;

CREATE TABLE TDB$VIEWS (
  NAME VARCHAR(32) PRIMARY KEY,
  ATSQL_QUERY LONG NOT NULL
);
```

Chapter 3

System Architecture

Tiger is a temporal database system front-end written in SWI Prolog. The Tiger front-end consists of thirteen modules which contain a total of about 4000 lines of (dense) code. Figure 3.1, depicts the module structure of Tiger. An arrow indicates the direction of an import (e.g., module 'meta' imports services from module 'unparser'). In the sequel, we briefly describe the functionality of the depicted modules.

tiger Definition of global data structures and variables.
Code size: 5KByte (source code)

interpret The main module acts as a dispatcher and controller. It receives the output from the parser (i.e., the parse tree) and determines the appropriate actions to be performed.
Code size: 10KByte (source code)

scanner Exports predicates to read input from standard IO and to scan text strings.
Code size: 10KByte (source code)

parser Parses a token list. Uses the definite clause grammar (DCG) notation of Prolog [CM87]. Standard parsing techniques from the literature [Wir86] have been employed to build a parse tree (i.e., *one symbol lookahead* (there are some exceptions) and *recursive descent*). Although backtracking is an inherent feature of Prolog and DCGs, it is used sparingly. Experience has shown that the (extensive) use of backtracking results in mediocre runtime performance [Bur92, p.15ff].
Code size: 18KByte (source code)

rewrite Normalizes ATSQL commands, e.g., determines missing table aliases. Performs also logical rewriting of commands to simplify/optimize the sub-

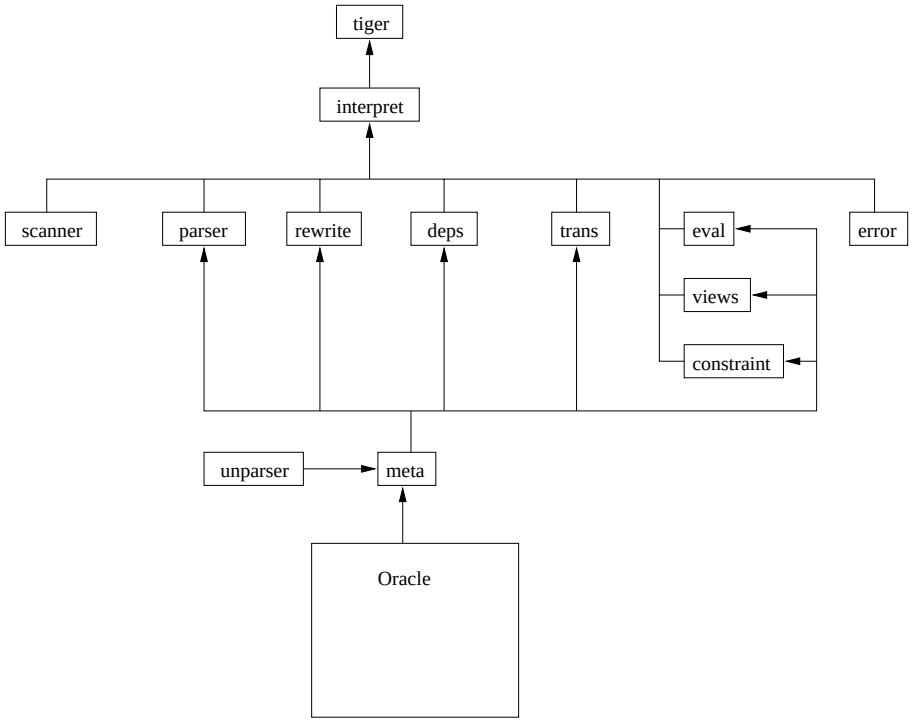

Figure 3.1: The architecture of Tiger

sequent translation.
Code size: 22KByte (source code)

deps Checks and handles dependencies implied by views, integrity constraints, and assertions.
Code size: 6KByte (source code)

trans Translates ATSQL queries to SQL queries. SQL queries are represented as Prolog structures so that they can be manipulated easier.
Code size: 9KByte (source code)

dml Handles ATSQL data manipulation statements. Particularly the handling of transaction time is done in this module.
Code size: 12KByte (source code) code)

views Handling of views. Right now limited to the migration of upward compatible views to temporal upward compatible views.

constraint Responsible for integrity constraints and assertions. Provides support for migrating upward compatible integrity constraints. Checks the consistency of the database.
Code size: 11KByte (source code)

meta Controls Oracle database accesses in general and access to the meta data of Tiger in particular. In its responsibility are different optimization strategies for meta data access. For example, parts of the meta data is cached in main memory. Also some commands are precompiled and associated with permanent cursors (see the cursor facility of Oracle [Nev86] for details).
Code size: 7KByte (source code)

unparser Generation of SQL strings. Responsible for the handling of periods.
Code size: 15KByte (source code)

Chapter 4

ATSQL

4.1 Data Model

Tiger is a bitemporal period-timestamped database system. At the conceptual level relations are extended with a valid time and transaction time dimension. At the physical level four additional attributes are added to each relation, transaction time start, transaction time end, valid time start, and valid time end.

Valid time denotes the time when a statement was, is or will be valid in the real world. Thus, valid time is always associated with a piece of information. This property distinguishes valid time from explicit time which is an entity on its own (e.g., a birth date). Whether to employ valid time or explicit time in order to model the time is usually a design decision. For example, if we have to model the life-time of persons, two basically different approaches are possible. 1) life-time can be represented as *valid time* of person entities or 2) life-time can be represented by extending the person entity with an *explicit* attribute that captures birth date and date of death. Case 2) is appropriate whenever we just want to store and retrieve temporal information, and maybe do a few simple computations, such as determining the age of a person. However, as soon as more sophisticated reasoning about time is required (e.g., searching persons who lived at the same time), valid time should be used to model life-time.

Transaction time denotes the time during which a statement is or was current to the database. Like valid time, transaction time is always associated with a piece of information. The time point when the information was inserted into the database is the transcation time start point whereas the time point when it was either deleted or superseded by an updated tuple is the transaction time end point. As a result, transaction time is not provided by the user. If a statement is inserted into the database, the start point is set to the insertion time (= the time when the enclosing transaction commits) whereas the end point is set to *now*. Similarly, when a statement is deleted (or updated) the transaction time

end point is changed from *now* to the deletion time.

4.2 Crucial concepts

This chapter is to some degree an excerpt of [BJ96]. Please consult this paper for further explanations and details.

4.2.1 Upward Compatibility (UC)

Perhaps the most important aspect of ensuring a smooth migration of application code is to guarantee that all code without modification will work with the new model, exactly with the same functionality as with the existing system. We define a data model to be *syntactically upward compatible* with another data model if all the data structures and legal query expressions of the latter model are contained in the former model. It then holds true that, if a model is syntactically upward compatible with a legacy data model, then all existing application code will remain syntactically correct. ATSQL is syntactically upward compatible with SQL-92. Thus, the bulk of legacy SQL-92 application code is not affected by the transition to ATSQL.

There is one unintended ramification of the above definition. Any temporal extension that includes new reserved keywords will violate upward compatibility. The reason is that legacy query language statements may have employed such keywords as identifiers. Under the semantics of the new model, such statements will be illegal. Reserved words are added in all temporal query languages, and it is impractical to exclude them. This also holds for non-temporal query languages. For example, SQL–92 added some 112 reserved keywords to the 115 reserved keywords of its predecessor, SQL–89[1]. One proposed solution is to use *quoted* identifiers in legacy code where identifiers conflict with new reserved keywords and in new code, to avoid future problems. In conclusion, to follow current practice and to avoid being overly restrictive, we consider upward compatibility to be satisfied even when new keywords are added in the extension.

4.2.2 Temporal Upward Compatibility (TUC)

While essential, upward compatibility is only a first step. Upon adopting a temporal data model, the benefits of the built-in temporal support are only realized incrementally, by modifying existing application code or developing new application code that exploits the temporal capabilities. A next step is thus to formulate requirements that aim at ensuring a harmonious coexistence of legacy application code and new, temporally-enhanced application code.

[1] Reference [MS93] provides a list of 10 items with incompatibilities among SQL–89 and SQL–92, with the keyword aspect being one item.

Note that there's no friction between existing code and new code if disjoint sets of relations are used. Rather, the potential problem occurs when new application code needs to use a combination of existing and new, temporal relations or needs to change existing relations to become temporal.

Based on this observation, temporal upward compatibility states that existing applications on snapshot relations must continue to work unmodified when the relations are altered to become temporal. Intuitively, the requirement is that a query q must return the same result on an associated snapshot database db as on the temporal counterpart of the database, $\mathcal{T}(db)$. Further, updates should not affect this.

Definition 1 (temporal upward compatibility) *Let a temporal and a snapshot data model be given by* $M_T = (DS_T, QL_T)$ *and* $M_S = (DS_S, QL_S)$*, respectively. Also, let* $\mathcal{T}$ *be an operator that changes the type of a snapshot relation to the temporal relation with the same explicit attributes. Next, let* $\mathcal{U} = u_1, u_2, \ldots, u_n$ $(n \geq 0)$ *denote a sequence of update operations. With these definitions, model* M_T *is* temporal upward compatible *with model* M_S *iff*

- M_T *is upward compatible with* M_S *and*
- $\forall db_S \in DS_S\ (\forall\, \mathcal{U}\ (\forall q_S \in QL_S\ (\langle\!\langle q_S(\mathcal{U}(db_S))\rangle\!\rangle_{M_S} = (\langle\!\langle q_S(\mathcal{U}(\mathcal{T}(db_S)))\rangle\!\rangle_{M_T}))))$.

4.2.3 Sequentiality (SEQ)

The sequentiality requirement aims at protecting the investments in programmer training and at ensuring continued efficient, cost-effective application development upon migration to a temporal model. This is achieved by exploiting the fact that programmers are likely to be comfortable with the non-temporal query language, e.g., SQL–92.

The requirement states that, for each query in SQL-92, ATSQL must offer a syntactically similar temporal query that is its "natural" generalization. With this requirement satisfied slightly modified versions of the SQL–92 queries on temporal relations are temporal queries with semantics that are easily ("naturally") understood in terms of the semantics of the the corresponding SQL–92 queries on snapshot relations. The familiarity of the similar syntax and "naturally" extended semantics is intended to make it possible for programmers to immediately and easily write a wide range of temporal queries, with little need for expensive training, few errors, and no significant initial drop in productivity.

An inherent part part of sequential statements is *snapshot reducibility*. We first define the notion of snapshot reducibility among query languages. We will use r and r^{bi} for denoting a snapshot and a bitemporal relation instance, respectively. Similarly, db and db^{bi} are sets of snapshot and bitemporal relation instances, respectively.

Snapshot reducibility implies that for all query expressions q in the snapshot model, there must exist a query q^{bi} in the temporal model so that for all db^{bi}

and for all time arguments, the commutativity diagram shown in Figure 4.1 holds.

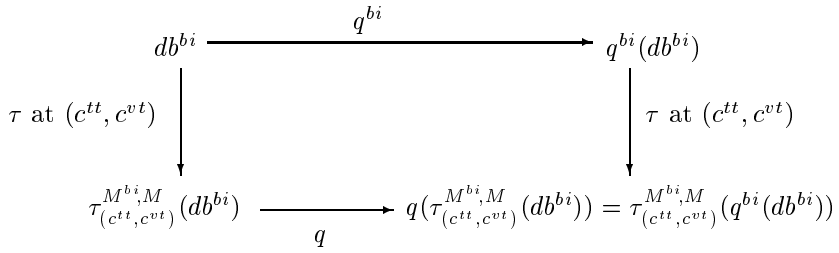

Figure 4.1: Snapshot reducibility of query q^{bi} with respect to query q

Observe that q^{bi} being snapshot reducible with respect to q poses no syntactical restrictions on q^{bi}. It is thus possible for q^{bi} to be quite different from q, and q^{bi} might be very involved. This is undesirable, as we would like the temporal model to be a straight-forward extension of the snapshot model. Consequently, we require that q^{bi} and q be syntactically similar. Specifically, we require that there exist two (possibly empty) strings, S_1 and S_2, such that each query q^{bi} in QL^{bi} that is snapshot reducible with respect to a query q in QL is syntactically identical to $S_1 q S_2$.

Figure 4.2 illustrates snapshot reducibility. The temporal query q' is snapshot reducible to the snapshot query q. Query q' is applied to the temporal relation (the sequence of states across the top of the figure) and results in a temporal relation, which is represented by the sequence of states across the bottom.

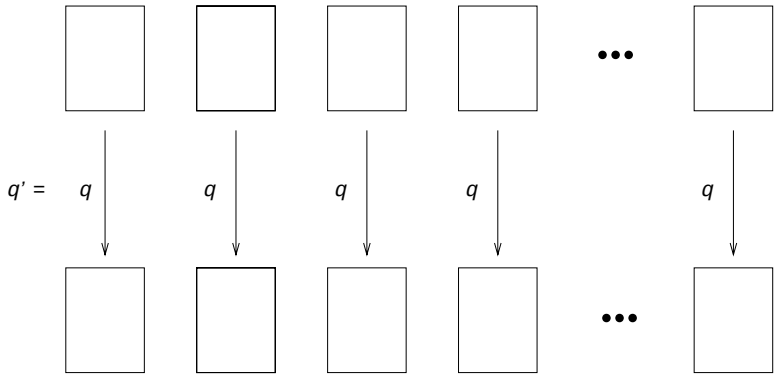

Figure 4.2: Evaluation of a Snapshot Reducible Temporal Query q' on a Temporal Relation

4.2.4 Nonsequentiality (NONSEQ)

In a sequential query, the information in a particular state of the resulting temporal relation is derived solely from information in the state at that same time of the source relation(s). However, there are many reasonable queries that cannot be expressed as sequenced queries. Such queries are illustrated in Figure 4.3, in which each state of the resulting relation requires information from possibly all states of the source relation. We term these queries *nonsequenced* ones. With these queries included, we have the full functionality that may be expected from a temporal query language.

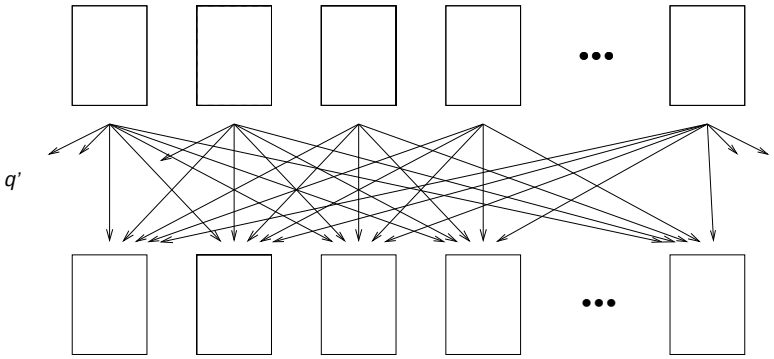

Figure 4.3: Evaluation of a nonsequenced temporal query q' on a temporal relation

In Figure 4.3, two temporal relations are shown, one consisting of the states across the top of the figure, and the other, the result of the query, consisting of the states across the bottom of the figure. A single query q performs the possibly complex computation, with the information usage illustrated by the downward pointing arrows. Whenever the computation of a single state of the result relation may utilize information from a state at a different time, that query is non-sequenced.

4.3 Syntax

4.3.1 Reserved Words

```
keyword("BEGIN",begin).
keyword("BEGINNING",beginning).
keyword("CONTAINS",con).
keyword("CURRENT",current).
keyword("DATE",date).
keyword("DAY",day).
keyword("NONSEQUENCED",nonsequenced).
keyword("NOW",now).
keyword("OVERLAPS",over).
keyword("PERIOD",period).
keyword("PRECEDES",pre).
keyword("SECOND",second).
```

```
keyword("END",end).
keyword("FIRST",first).
keyword("FOREVER",forever).
keyword("HOUR",hour).
keyword("INTERVAL",interval).
keyword("LAST",last).
keyword("MEETS",mee).
keyword("MINUTE",minute).
keyword("MONTH",month).
keyword("SEQUENCED",sequenced).
keyword("SYSDATE",sysdate).
keyword("TIMESTAMP",timestamp).
keyword("TO",to).
keyword("TRANSACTION",transaction).
keyword("TTIME",ttime).
keyword("VALID",valid).
keyword("VTIME",vtime).
keyword("YEAR",year).
```

4.3.2 Temporal Built-Ins

Most built-ins were chosen to be consistent with TSQL2 and/or the SQL standard.

We first illustrate the syntax of periods.

```
PERIOD '1992/4/21 - 1997/5/31'
PERIOD '1992/4/21 - NOW'
PERIOD 'CURRENT - 1999/5/17'
PERIOD 'CURRENT - NOW'
PERIOD '1995/4/21'
```

Periods are assumed to be closed. `CURRENT` is a shorthand for the current time and is replaced whereas `NOW` denotes a moving point on the time line. The last example is a shorthand for `PERIOD '1995/4/21-1995/4/21'`. In the future there will be support for different granularities and multiple timestamp formats.

We continue with a list of built-in functions to access and manipulate periods.

- `VTIME(`r`)` returns the valid time of relation r
- `VTIME(`r`)` returns the transaction time of relation r
- `BEGIN(`P`)` returns the start point of a period
- `END(`P`)` returns the end point of a period
- `FIRST(`TP_1, TP_2`)` returns the earlier of two time points
- `LAST(`TP_1, TP_2`)` returns the later of two time points
- `DURATION(`$P, Gran$`)` returns the duration of a period at the specified granularity

Finally, a set of built-in temporal predicates allows to conveniently relate periods. Their semantics is given of terms of conventional relationships on start and end points.

- P_1 `=` P_2 iff $P_1^- = P_2^-$ and $P_1^+ = P_2^+$

- P_1 `CONTAINS` P_2 iff $P_1^- \leq P_2^-$ and $P_1^+ \geq P_2^+$
- P_1 `MEETS` P_2 iff $succ(P_1^+) = P_2^-$
- P_1 `OVERLAPS` P_2 iff $P_1^- \leq P_2^+$ and $P_2^- \leq P_1^+$
- P_1 `PRECEDES` P_2 iff $P_1^+ < P_2^-$

4.3.3 Flags

We now discuss the syntactic constructs of ATSQL to achieve UC, TUC, SEQ, and NONSEQ. We provide an EBNF syntax for each of our specific extensions to SQL–92. We thus focus on the temporal extensions and gloss over the details of SQL–92. In addition, we will focus on queries, which are the most complex statements, but will also consider other statements. In the EBNF productions that follow, terminals are of the form `"xxx"`, i.e., enclosed in quotation marks. Non-terminals of the form `<xxx>` derive from the SQL–92 standard [MS93, p.481ff], and new non-terminals are of the form `xxx` (without quotation marks). Omitting these new non-terminals yields the original (slightly simplified) SQL–92 productions.

- In *queries* and cursor expressions (termed `<cursor specification>` in SQL–92) the flags are placed at the outermost level, right at the beginning.

```
<cursor specification> ::=
   flags <query expression> [ <order by clause> ] |
   "(" flags <query expression> ")" coal
      [ <order by clause> ]
```

  The scope of the semantics implied by the flags is all parts of the query (e.g., including nested queries), with the exception of derived table expressions in the from clause. The non-terminal `coal` is used for specifying coalescing, to be discussed later in this section.

- In *views*, the flags are placed immediately following the `AS` keyword.

```
<view definition> ::=
   "CREATE" "VIEW" <table_name>
   [ "(" <view column list> ")" ] "AS"
   ( flags <query expression> |
     "(" flags <query expression> ")" coal )
```

- Flags can be associated with *derived table expressions* in from clauses. The motivation is that derived tables may be meaningfully computed independently of the computation of the remainder of the containing query. Put differently, derived table expressions have their own scope and may be replaced by views or auxiliary tables. This presents an opportunity to allow derived tables expressions to have their own individual flags. This adds flexibility to the language and improves its usefulness. As a syntactic shorthand, coalescing is also allowed after table names in the from clause (in order to facilitate point-based queries).

```
<table reference> ::=
   <table name> coal [ [ "AS" ] <correlation name> ] |
   "(" flags <query expression> ")" coal [ "AS" ]
      <correlation name>
```

  Note that, while syntactically similar, derived tables in the from clause are quite different from subqueries in the where clause. Subqueries can be correlated with the main query and cannot be evaluated independently.

- In an *assertion*, the flags are placed right after the `CHECK` keyword.

```
<assertion definition> ::=
   "CREATE" "ASSERTION" <constraint Name>
   CHECK flags "(" <search condition> ")"
```

- *Table and column constraints* are syntactic shorthands for assertions. The flags are placed right in front of the table and column constraints, respectively.

```
<column definition> ::=
   <column name> flags <column constraint definition>

<table constraint definition> ::=
   <constraint name definition> flags <table constraint>
```

- As with queries, the flags are placed in front of *modification statements*.

```
<SQL data change statement> ::=
   flags <insert statement> |
   flags <delete statement> |
   flags <update statement>
```

Summarizing, flags are associated with all "statements" that can be evaluated meaningfully on their own. Examples include queries, data manipulation statements, assertions, integrity constraints, and views. In general, flags are placed in front of statements to emphasize their impact upon the entire statement.

Before we exemplify the different statement classes with a guided tour, we continue with an EBNF syntax for the flags and discuss their meaning.

```
flags     ::= [ flag [ "AND" flag ] ] [ "SET" "VALID" vt_range ]
flag      ::= [ modifier ] dimension [ domain ]
modifier  ::= "SEQUENCED" | "NONSEQUENCED"
dimension ::= "TRANSACTION" | "VALID"
domain    ::= period_constant
vt_range  ::= period_expression
```

The meaning of the flags naturally divides into four orthogonal parts, namely the specification of the core semantics, the time-domain specification, the time-range specification, and specification of coalescing. We discuss each in turn.

The following three types of flags determine the *core semantics* of ATSQL statements. Each of the three types of flags apply independently to valid time and transaction time.

<*empty flag*> A missing flag for a time dimension (i.e., valid or transaction) dictates upward compatibility (UC) when neither of the underlying argument relations support that time; otherwise, evaluation according to temporal upward compatibility (TUC) is dictated. For queries, the time dimension will not be present in the result relation.

`SEQUENCED` When this keyword is present for a time dimension, evaluation consistent with sequenced semantics (SEQ), i.e., built-in timestamp-related processing, is dictated for the time dimension. The time dimension will be present in relations that result from queries.

`NONSEQUENCED` This keyword signals nonsequenced semantics (NONSEQ), i.e., timestamp processing that is controlled by the application rather than the temporal DBMS. The affected time dimension is not present in query results (with this flag, the time effectively becomes an explicit attribute that can be included in the result similarly to how other explicit attributes are included).

With two time dimensions, the three cases lead to a total of nine kinds of statements, as summarized in Table 4.1. For simplicity, we have omitted permutations of the valid and transaction time flag, and we abbreviate `VALID` by `VT`, `TRANSACTION` by `TT`, `SEQUENCED` by `SEQ`, and `NONSEQUENCED` by `NS`.

The next step is to add time-domain and time-range specifications. The *time domain* is a period constant that may be placed right after the `VALID` and `TRANSACTION` keywords, respectively. It restricts the database to the part that

syntax	semantics	
	vt	tt
<SQL–92>	(T)UC	(T)UC
`SEQ VT` <SQL–92>	SEQ	(T)UC
`NS VT` <SQL–92>	NONSEQ	(T)UC
`SEQ TT` <SQL–92>	(T)UC	SEQ
`NS TT` <SQL–92>	(T)UC	NONSEQ
`SEQ VT AND SEQ TT` <SQL–92>	SEQ	SEQ
`SEQ VT AND NS TT` <SQL–92>	SEQ	NONSEQ
`NS VT AND SEQ TT` <SQL–92>	NONSEQ	SEQ
`NS VT AND NS TT` <SQL–92>	NONSEQ	NONSEQ

Table 4.1: The Basic Usage of Flags in ATSQL

is valid or current during the respective period. A domain restriction is applied prior to the evaluation of a statement, i.e., in a preprocessing step.

For valid time, it can be meaningful to specify the valid time of the result, i.e., the *time range*. The `SET VALID` clause is used for this purpose. Note that it makes no sense to provide a similar clause for transaction time. Transaction-time semantics forbids this kind of user interaction [SA85]. The time range is set in a postprocessing step, i.e., after the evaluation of a query. Examples of time-domain and time-range specifications will be given in the guided tour that follows.

Finally, coalescing merges tuples with overlapping or adjacent timestamps, and identical corresponding attribute values (termed value equivalent), into a single tuple. Coalescing is allowed at the levels where the flags are also allowed. In addition, as a syntactic shorthand, a coalescing operation is permitted directly after a relation name in the from clause. In this case a coalesced instance of the relation, rather than the uncoalesced one, is considered.

```
coal ::= { "(" dimension ")" }
dimension ::= "valid" | "transaction"
```

The semantics of coalescing depends on the type of relation it is applied to. A snapshot relation cannot be coalesced. A valid-time relation can be coalesced in valid time only, and the equivalent is true for transaction-time relations. With a single time dimension, coalescing degenerates to a merging of value-equivalent tuples with overlapping or adjacent time period [BSS96]. With bitemporal relations the semantics is more subtle. Here, overlapping or adjacent time regions (rectangles) of value-equivalent tuples have to be merged. In the general case, overlapping rectangles do not coalesce into a single rectangle, which means that several result tuples have to be generated. This can be done in two ways: with the resulting rectangles maximized in valid time or in transaction time. We

use (VALID) for the former and (TRANSACTION) for the latter. These two basic operations can be combined to (TRANSACTION)(VALID), which means that we first coalesce in transaction time and then in valid time. As exemplified by the last two pictures, the sequence of coalescing operations matters. The sequence (TRANSACTION)(VALID) results in maximal valid-time periods, whereas (VALID)(TRANSACTION) results in maximal transaction-time periods.

4.4 Semantics

We define the semantics of ATSQL in terms of a mapping to standard and temporal relational algebra, both of which are defined here. To avoid the tedious complications related to duplicates, which have been explored in the past, we assume a set-based framework. This yields a concise coverage where the novel aspects of the general approach stand out more clearly. However, we emphasize that ATSQL follows the data model of SQL–92 and is thus not set-based.

The translation to (temporal) relational algebra expressions consists of two parts. First, we consider constructs at the level of functions and predicates. This step is straightforward and is discussed in the first section. The translation at the statement level, i.e., the translation of statements enhanced with flags, is much more involved (and important!). It is discussed in the subsequent three sections.

Constructs for Timestamp Manipulation

Temporal query languages generally define a variety of constructs to manipulate their various timestamp types. These include constructors (to create instances of the timestamp types), extractors (to extract constituent parts from timestamps), predicates (boolean-valued, for comparison), and operations (to create new timestamps from existing ones). Many constructs exist in the literature [Sno95, pp. 251–291]. They are relatively easy to define, and adding one more construct to a language has only a localized effect on the language design. Therefore, we only define a relatively small number of constructs here.

We will assume the timestamp representation adopted in Tiger where four TIMESTAMP attributes (VT$S, VT$E, TT$S, and TT$E, denoting valid time start, valid time end, transaction time start, and transaction time end, respectively) are used to represent valid and transaction time. This representation leads to the definitions given in Table 4.2, where *tp* and *iv*, possibly indexed, denote a time point of type TIMESTAMP and a time duration of type INTERVAL, respectively. Also, *per* is a shorthand for PERIOD '$tp_1 - tp_2$' and *granule* $\in$ {YEAR, MONTH, WEEK, DAY, HOUR, MINUTE, SECOND} denotes a granularity. The constructs VTIME and TTIME that extract timestamps return errors if the tuple variables they are applied to do not support valid time and transaction time, respectively. Note also that INTERSECT returns an illegal period if the

ATSQL	*Semantics*
$[\![$ PERIOD '$tp_1 - tp_2$' $]\!]_{ATSQL}$	TIMESTAMP'tp_1', TIMESTAMP'tp_2'
$[\![$ FIRST(TIMESTAMP'tp_1', TIMESTAMP'tp_2') $]\!]_{ATSQL}$	$min(tp_1, tp_2)$
$[\![$ LAST(TIMESTAMP'tp_1', TIMESTAMP'tp_2') $]\!]_{ATSQL}$	$max(tp_1, tp_2)$
$[\![$ VTIME(r) $]\!]_{ATSQL}$	TIMESTAMP'r.VT$S', TIMESTAMP'$r$.VT$E'
$[\![$ TTIME(r) $]\!]_{ATSQL}$	TIMESTAMP'r.TT$S', TIMESTAMP'$r$.TT$E'
$[\![$ BEGIN(per) $]\!]_{ATSQL}$	$[\![$ FIRST($[\![per]\!]_{ATSQL}$) $]\!]_{ATSQL}$
$[\![$ END(per) $]\!]_{ATSQL}$	$[\![$ LAST($[\![per]\!]_{ATSQL}$) $]\!]_{ATSQL}$
$[\![per_1$ PRECEDES $per_2]\!]_{ATSQL}$	$[\![$ END(per_1) $]\!]_{ATSQL} < [\![$ BEGIN(per_2) $]\!]_{ATSQL}$
$[\![per_1$ MEETS $per_2]\!]_{ATSQL}$	$[\![$ END(per_1) $]\!]_{ATSQL} = [\![$ BEGIN(per_2) $-^{granule} 1]\!]_{ATSQL}$
$[\![per_1$ OVERLAPS $per_2]\!]_{ATSQL}$	$[\![$ END(per_1) $]\!]_{ATSQL} \geq [\![$ BEGIN(per_2) $]\!]_{ATSQL} \wedge$ $[\![$ END(per_2) $]\!]_{ATSQL} \geq [\![$ BEGIN(per_1) $]\!]_{ATSQL}$
$[\![per_1$ CONTAINS $per_2]\!]_{ATSQL}$	$[\![$ BEGIN(per_2) $]\!]_{ATSQL} \geq [\![$ BEGIN(per_1) $]\!]_{ATSQL} \wedge$ $[\![$ END(per_2) $]\!]_{ATSQL} \leq [\![$ END(per_1) $]\!]_{ATSQL}$
$[\![per +$ INTERVAL'iv' $]\!]_{ATSQL}$	$[\![$ BEGIN(per) $]\!]_{ATSQL} + iv,$ $[\![$ END(per) $]\!]_{ATSQL} + iv$
$[\![$ INTERSECT(per_1, per_2) $]\!]_{ATSQL}$	$max([\![$ BEGIN(per_1) $]\!]_{ATSQL}, [\![$ BEGIN(per_2) $]\!]_{ATSQL}),$ $min([\![$ END(per_1) $]\!]_{ATSQL}, [\![$ END(per_2) $]\!]_{ATSQL})$
$[\![$ DURATION($per, granule$) $]\!]_{ATSQL}$	$[\![$ END(per) $]\!]_{ATSQL} -^{granule} [\![$ BEGIN(per) $]\!]_{ATSQL})$

Table 4.2: Definition of Simple ATSQL Constructs

two argument periods do not overlap. We will use the constructs defined in Table 4.2 throughout, including in relational algebra expressions, e.g., in selection predicates. This makes the expressions more readable. It is straightforward to adapt these definitions to different representations, e.g., a representation that is based on the `PERIOD` data type of the evolving part SQL/Temporal of the SQL3 standard.

Query Expressions

Recall that we define the meaning of ATSQL query expressions by translating them to well-defined algebraic expressions. As a precursor, we introduce the notation that we will use in the algebra expressions.

We use $\langle t \rangle$, $\langle t \| VT \rangle$, $\langle t \| TT \rangle$, and $\langle t \| VT, TT \rangle$ to denote tuple variables ranging over snapshot, valid-time, transaction-time, and bitemporal relations, respectively. The vertical double-bar "$\|$" is used to separate the explicit attributes from the implicit timestamps. The valid time is referred to as VT, the transaction time as TT.

In the definitions, we need auxiliary operators that timeslice relations and turn timestamps into regular, explicit attributes. These operators are overloaded to apply to valid-time, transaction-time, and bitemporal relations, and they have variants for both valid and transaction time. There are two timeslice operations. The first, τ_{tp}, selects all tuples in the argument relation with a timestamp that overlaps time point *tp*. The time dimension used in this selection is not present in the result relation. The second timeslice operation, δ_{per}, returns all argument tuples that overlap with period *per*. The timestamp of a result tuple is the intersection of *per* with the tuple's original timestamp. The snapshot operation SN turns a time dimension into an explicit attribute. Note that SN is not needed at the implementation level, where all attributes are explicit (cf. Section 4.4).

With these conventions in place, Table 4.3 gives the semantics for core ATSQL statements (cf. Table 4.1). In the table, $[\![$<SQL–92>$]\!]_{SQL-92}$ evaluates to the standard relational algebra expression that corresponds to <SQL–92> [?, GT91]. Next, $[\![$<SQL–92>$]\!]_T$, where $T \in \{vt, tt, bi\}$, evaluates to the same algebraic expression as does $[\![$<SQL–92>$]\!]_{SQL-92}$, except that every nontemporal relational algebra operator (e.g., $\times, \sigma, \pi$) is replaced by the corresponding temporal relational algebra operator (e.g., $\times^T, \sigma^T, \pi^T$). The algebras are defined in Section ??. The following two examples illustrate the definition.

Example 1 *The ATSQL query, Q_1, below is an example of a non-sequenced query. The argument relations are assumed to be bitemporal.*

```
NONSEQUENCED VALID
  SELECT p.X
  FROM p, q
  WHERE p.X = q.X
```

$$[\![\text{<SQL–92>}]\!]_{ATSQL}(r_1,\ldots,r_n) \triangleq$$
$$\quad [\![\text{<SQL–92>}]\!]_{SQL-92}(\tau^{tt}_{now}(\tau^{vt}_{now}(r_1)),\ldots,\tau^{tt}_{now}(\tau^{vt}_{now}(r_n)))$$

$$[\![\texttt{SEQ VT } \text{<SQL–92>}]\!]_{ATSQL}(r_1,\ldots,r_n) \triangleq$$
$$\quad [\![\text{<SQL–92>}]\!]_{vt}(\tau^{tt}_{now}(r_1),\ldots,\tau^{tt}_{now}(r_n))$$

$$[\![\texttt{NS VT } \text{<SQL–92>}]\!]_{ATSQL}(r_1,\ldots,r_n) \triangleq$$
$$\quad [\![\text{<SQL–92>}]\!]_{SQL-92}(\tau^{tt}_{now}(SN^{vt}(r_1)),\ldots,\tau^{tt}_{now}(SN^{vt}(r_n)))$$

$$[\![\texttt{SEQ TT } \text{<SQL–92>}]\!]_{ATSQL}(r_1,\ldots,r_n) \triangleq$$
$$\quad [\![\text{<SQL–92>}]\!]_{tt}(\tau^{vt}_{now}(r_1),\ldots,\tau^{vt}_{now}(r_n))$$

$$[\![\texttt{NS TT } \text{<SQL–92>}]\!]_{ATSQL}(r_1,\ldots,r_n) \triangleq$$
$$\quad [\![\text{<SQL–92>}]\!]_{SQL-92}(\tau^{vt}_{now}(SN^{tt}(r_1)),\ldots,\tau^{vt}_{now}(SN^{tt}(r_n)))$$

$$[\![\texttt{SEQ VT AND SEQ TT } \text{<SQL–92>}]\!]_{ATSQL}(r_1,\ldots,r_n) \triangleq$$
$$\quad [\![\text{<SQL–92>}]\!]_{bi}(r_1,\ldots,r_n)$$

$$[\![\texttt{SEQ VT AND NS TT } \text{<SQL–92>}]\!]_{ATSQL}(r_1,\ldots,r_n) \triangleq$$
$$\quad [\![\text{<SQL–92>}]\!]_{vt}(SN^{tt}(r_1),\ldots,SN^{tt}(r_n))$$

$$[\![\texttt{NS VT AND SEQ TT } \text{<SQL–92>}]\!]_{ATSQL}(r_1,\ldots,r_n) \triangleq$$
$$\quad [\![\text{<SQL–92>}]\!]_{tt}(SN^{vt}(r_1),\ldots,SN^{vt}(r_n))$$

$$[\![\texttt{NS VT AND NS TT } \text{<SQL–92>}]\!]_{ATSQL}(r_1,\ldots,r_n) \triangleq$$
$$\quad [\![\text{<SQL–92>}]\!]_{SQL-92}(SN^{tt}(SN^{vt}(r_1)),\ldots,SN^{tt}(SN^{vt}(r_n)))$$

Table 4.3: Semantics of Core ATSQL Queries

```
AND VTIME(p) PRECEDES VTIME(q)
```

This query is defined by the relational algebra expression given next.

$$[\![Q_1]\!]_{ATSQL}(p,q) =$$
$$\pi_{p.X}(\sigma_{p.X=q.X}(\sigma_{VTIME(p)\ PRECEDES\ VTIME(q)}(SN^{vt}(\tau^{tt}_{now}(p)) \times SN^{vt}(\tau^{tt}_{now}(q)))))$$

Note that the mapping from SQL–92 queries to relational algebra is still the same. The temporal selection condition can be viewed as a syntactic shorthand for a standard selection condition (cf. Table 4.2). The only addition is the "adjustment" of the relations (SN^{vt} and τ^{tt}_{now}) to fit the non-sequenced evaluation mode in valid time dimension and the temporal upward compatible evaluation mode in transaction time dimension. ∎

Example 2 *The following ATSQL query, termed Q_2, is sequenced in both valid and transaction time.*

```
SEQUENCED VALID AND SEQUENCED TRANSACTION
  SELECT p.X
  FROM p, q
  WHERE p.X = q.X
```

It is defined by the following temporal relational algebra expression.

$$[\![Q_2]\!]_{ATSQL}(p,q) = \pi^{bi}_{p.X}(\sigma^{bi}_{p.X=q.X}(p \times^{bi} q))$$

Apart from the superscripts, which are added to the relational algebra operators, the translation between SQL–92 queries and relational algebra expressions has not changed at all. ∎

Domain and Range Specifications

Next, we define the semantics of domain and range specifications. A time-domain restriction restricts the argument relations in a query to contain only tuples that are valid during a specific period. Thus, only the parts of argument tuples that intersect with the time-domain restriction are considered when the query is evaluated. This is formalized in Table 4.4.

Next, we can also specify a time range, using the flag "`SET VALID` <range>" where <range> is period valued, that determines the valid times of the result tuples. There are two different situations. First, if the core statement is a `SEQUENCED VALID` statement then the automatically computed valid time is replaced by the value resulting from evaluating the time-range specification. Second, for all other core statements, prepending `SET VALID` <range> results in the inclusion of valid time into the result. Because these core statements return results that do not contain valid-time timestamps, the type of the result is changed. The valid time of a tuple is that resulting from evaluating <range>. The details are given in Table 4.5.

$$[\![\text{<modifier> } \texttt{VALID} \text{ <domain> <ATSQL>}]\!]_{ATSQL}(r_1, \ldots, r_n) \triangleq$$
$$[\![\text{<modifier> } \texttt{VALID} \text{ <ATSQL>}]\!]_{ATSQL}(\delta^{vt}_{\text{<domain>}}(r_1), \ldots, \delta^{vt}_{\text{<domain>}}(r_n))$$

$$[\![\text{<modifier> } \texttt{TRANSACTION} \text{ <domain> <ATSQL>}]\!]_{ATSQL}(r_1, \ldots, r_n) \triangleq$$
$$[\![\text{<modifier> } \texttt{TRANSACTION} \text{ <ATSQL>}]\!]_{ATSQL}(\delta^{tt}_{\text{<domain>}}(r_1), \ldots, \delta^{tt}_{\text{<domain>}}(r_n))$$

Table 4.4: Definition of ATSQL Domain Restrictions

$$[\![\texttt{SET VALID} \text{ <range><ATSQL>}]\!]_{ATSQL}(r_1, \ldots, r_n) \triangleq$$

$$\begin{cases}
\{\langle t \| VT \rangle \mid \langle t \rangle \in [\![\text{<ATSQL>}]\!]_{ATSQL}(r_1, \ldots, r_n) \wedge \\
\quad VT = \text{<range>}(t)\} \\
\quad\quad \text{if } [\![\text{<ATSQL>}]\!]_{ATSQL}(r_1, \ldots, r_n) \\
\quad\quad\quad \text{evaluates to a snapshot relation} \\
\{\langle t \| VT \rangle \mid \langle t \| VT' \rangle \in [\![\text{<ATSQL>}]\!]_{ATSQL}(r_1, \ldots, r_n) \wedge \\
\quad VT = \text{<range>}(t)\} \\
\quad\quad \text{if } [\![\text{<ATSQL>}]\!]_{ATSQL}(r_1, \ldots, r_n) \\
\quad\quad\quad \text{evaluates to a valid-time relation} \\
\{\langle t \| VT, TT \rangle \mid \langle t \| TT \rangle \in [\![\text{<ATSQL>}]\!]_{ATSQL}(r_1, \ldots, r_n) \wedge \\
\quad VT = \text{<range>}(t)\} \\
\quad\quad \text{if } [\![\text{<ATSQL>}]\!]_{ATSQL}(r_1, \ldots, r_n) \\
\quad\quad\quad \text{evaluates to a transaction-time relation} \\
\{\langle t \| VT, TT \rangle \mid \langle t \| VT', TT \rangle \in [\![\text{<ATSQL>}]\!]_{ATSQL}(r_1, \ldots, r_n) \wedge \\
\quad VT = \text{<range>}(t)\} \\
\quad\quad \text{if } [\![\text{<ATSQL>}]\!]_{ATSQL}(r_1, \ldots, r_n) \\
\quad\quad\quad \text{evaluates to a bitemporal relation}
\end{cases}$$

Table 4.5: Definition of ATSQL Range Specifications

Coalescing

Any ATSQL query that returns a temporal relation may be coalesced. To define coalescing, let <ATSQL> denote any ATSQL query. If this query returns a valid-time relation, it may be modified to `(`<ATSQL>`)(VALID)`, to return the coalesced version of the valid-time relation. The obvious corresponding result holds when replacing valid time by transaction time. If the query returns a bitemporal relation, it may be coalesced in valid time, in transaction time, or in a combination of the two. Table **??** provides the definitions. Definitions of representative versions of operator *coal* will be given shortly.

$$[\![\text{<ATSQL>}\texttt{(VALID)}]\!]_{ATSQL}(r_1,\ldots,r_n) \triangleq$$

$$\begin{cases} coal^{vt}([\![\text{<ATSQL>}]\!]_{ATSQL}(r_1,\ldots,r_n)) \\ \qquad \text{if } [\![\text{<ATSQL>}]\!]_{ATSQL}(r_1,\ldots,r_n) \\ \qquad\quad \text{evaluates to a valid-time relation} \\ coal^{bi}_{vt}([\![\text{<ATSQL>}]\!]_{ATSQL}(r_1,\ldots,r_n)) \\ \qquad \text{if } [\![\text{<ATSQL>}]\!]_{ATSQL}(r_1,\ldots,r_n) \\ \qquad\quad \text{evaluates to a bitemporal relation} \end{cases}$$

$$[\![\text{<ATSQL>}\texttt{(TRANSACTION)}]\!]_{ATSQL}(r_1,\ldots,r_n) \triangleq$$

$$\begin{cases} coal^{tt}([\![\text{<ATSQL>}]\!]_{ATSQL}(r_1,\ldots,r_n)) \\ \qquad \text{if } [\![\text{<ATSQL>}]\!]_{ATSQL}(r_1,\ldots,r_n) \\ \qquad\quad \text{evaluates to a transaction-time relation} \\ coal^{bi}_{tt}([\![\text{<ATSQL>}]\!]_{ATSQL}(r_1,\ldots,r_n)) \\ \qquad \text{if } [\![\text{<ATSQL>}]\!]_{ATSQL}(r_1,\ldots,r_n) \\ \qquad\quad \text{evaluates to a bitemporal relation} \end{cases}$$

Chapter 5

Oracle MetaDB Tables

For a comprehensiove description of the Oracle meta database please consult Oracle documentation. Here we just provide the descriptions of those table that help you to get started and those tables/views that are used explicitly by Tiger.

```
SQL> desc cat;
 Name                           Null?    Type
 ------------------------------ -------- ----
 TABLE_NAME                     NOT NULL VARCHAR2(30)
 TABLE_TYPE                              VARCHAR2(11)

SQL> desc user_tab_columns;
 Name                           Null?    Type
 ------------------------------ -------- ----
 TABLE_NAME                     NOT NULL VARCHAR2(30)
 COLUMN_NAME                    NOT NULL VARCHAR2(30)
 DATA_TYPE                               VARCHAR2(9)
 DATA_LENGTH                    NOT NULL NUMBER
 DATA_PRECISION                          NUMBER
 DATA_SCALE                              NUMBER
 NULLABLE                                VARCHAR2(1)
 COLUMN_ID                      NOT NULL NUMBER
 DEFAULT_LENGTH                          NUMBER
 DATA_DEFAULT                            LONG
 NUM_DISTINCT                            NUMBER
 LOW_VALUE                               RAW(32)
 HIGH_VALUE                              RAW(32)
 DENSITY                                 NUMBER
```

```
SQL> desc user_constraints
 Name                           Null?    Type
 ------------------------------ -------- ----
 OWNER                          NOT NULL VARCHAR2(30)
 CONSTRAINT_NAME                NOT NULL VARCHAR2(30)
 CONSTRAINT_TYPE                         VARCHAR2(1)
 TABLE_NAME                     NOT NULL VARCHAR2(30)
 SEARCH_CONDITION                        LONG
 R_OWNER                                 VARCHAR2(30)
 R_CONSTRAINT_NAME                       VARCHAR2(30)
 DELETE_RULE                             VARCHAR2(9)
 STATUS                                  VARCHAR2(8)

SQL> desc user_cons_columns
 Name                           Null?    Type
 ------------------------------ -------- ----
 OWNER                          NOT NULL VARCHAR2(30)
 CONSTRAINT_NAME                NOT NULL VARCHAR2(30)
 TABLE_NAME                     NOT NULL VARCHAR2(30)
 COLUMN_NAME                    NOT NULL VARCHAR2(30)
 POSITION                                NUMBER

SQL> desc user_dependencies
 Name                           Null?    Type
 ------------------------------ -------- ----
 NAME                           NOT NULL VARCHAR2(30)
 TYPE                                    VARCHAR2(12)
 REFERENCED_OWNER                        VARCHAR2(30)
 REFERENCED_NAME                NOT NULL VARCHAR2(30)
 REFERENCED_TYPE                         VARCHAR2(12)
 REFERENCED_LINK_NAME                    VARCHAR2(128)

SQL> desc user_views
 Name                           Null?    Type
 ------------------------------ -------- ----
 VIEW_NAME                      NOT NULL VARCHAR2(30)
 TEXT_LENGTH                             NUMBER
 TEXT                                    LONG
```

Chapter 6

Command Overview

This chapter provides a quick tour through Tiger commands. First, miscellaneous commands, e.g., commands to open and close a database and commands to query meta information, are discussed. Then a guided tour is presented that illustrates many of the core features of Tiger.

The sections is kept informal. Usually we put one or several examples to illustrate a command. A formal syntax can be found in chapter **??**.

6.1 Miscellaneous Commands

open The command

```
open 'scott/tiger';
```

connects to the database *scott* with password *tiger*. If you connect through the world wide web you do not have to worry about databases and passwords—they are handled transparantly.

verbose The command

```
verbose on;
```

switches to verbose mode. This means that the generated SQL statements will be displayed on the screen. (Usually, only a dot appears on the screen instead of the actual SQL statement.) Switch the verbose mode off by typing

```
verbose off;
```

check The

```
check;
```

checks the consistency of the database.

close The command

```
close;
```

closes the database you are currently connected to. Note that it makes no sense to use this command if you do not have an account and a password.

6.2 ATSQL Commands

This section provides the complete guided tour that illustrates various aspects of ATSQL. The guided tour corresponds to the one that is available online. Additional explanations are given where appropriate.

```
/* First it is a good idea to check out the content of the DB.
   In Oracle you do so by executing the following command.
*/
SELECT * FROM cat;
/* See the Oracle documentation for more information about
   Oracle's meta tables.  This command should return a few
   object names starting with 'TDB$'.  They are meta table used
   by Tiger.  If they are not there you cannot proceed.
*/

/* If someone else has been running the guided tour before it
   is likely that some objects are left which are created during
   the guided tour.  Issue the following sequence to drop them.
*/
DROP ASSERTION not_prozac_and_morphine;
DROP VIEW addicted;
DROP TABLE drug;
DROP TABLE prescription;
DROP TABLE patient;
SELECT * FROM cat;
```

First, we employ upward compatible statements to define a small example database and to populate it. Note that the concept of upward compatibility ensures that *all* legacy Oracle-SQL statements keep being valid. Feel free to add your own statements or to modify the given ones. Whatever is a legal Oracle statement is also a legal Tiger statement.

```
CREATE TABLE patient (
  Id        INTEGER PRIMARY KEY,
  Name      VARCHAR(32),
  Address   VARCHAR(90),
  Birthdate DATE );
```

```
CREATE TABLE prescription(
  Patient_Id INTEGER REFERENCES patient(Id),
  Drug_Id    INTEGER,
  Doctor     VARCHAR(32),
  Issue_Date DATE );

INSERT INTO patient VALUES (
  5596544,
  'Bob Marley',
  'Miami',
  '1945/02/06' );
INSERT INTO patient VALUES (
  1846549,
  'Jim Morison',
  'Paris',
  '1943/12/08' );
INSERT INTO patient VALUES (
  9734859,
  'Jerry Garcia',
  'Forest Knolls',
  '1942/08/01' );
INSERT INTO patient VALUES (
  1734654,
  'Janis Joplin',
  'San Francisco',
  '1943/01/19' );

INSERT INTO prescription VALUES (
  5596544, 45477, 'Dr Quincy', '1992/7/22' );
INSERT INTO prescription VALUES (
  5596544, 17575, 'Dr Hook', '1992/8/15' );
INSERT INTO prescription VALUES (
  1734654, 16584, 'Dr Jekyll', '1991/12/25' );

SELECT * FROM patient;
SELECT * FROM prescription;

CREATE VIEW addicted (Patient_Name, Num_Of_Prescriptions) AS
  SELECT Name, COUNT(*)
  FROM patient, prescription
  WHERE patient.Id = prescription.Patient_Id
  GROUP BY Name
  HAVING COUNT(*) > 1;
```

```
SELECT * from addicted;
```

Next we employ temporally upward compatible (TUC) statements. Syntactically, TUC statements are identical to UC statements (apart from an extended `alter` statement which is used to add a time dimension to a table).

Another thing noteworthy is the `CHANGE SYSDATE` clause that is used below. This clause permits you to set the system time. Of course in a production system such a clause is not available (at least not to regular users). It is only provided for your convenience so that you can make your input independent of the actual time. If you don't like this feature simply omit the `CHANGE SYSDATE` clause. In this case Tiger will automatically choose the actual time.

```
CHANGE SYSDATE TO '1992/08/15';
ALTER TABLE patient ADD VALID;
ALTER TABLE prescription ADD VALID;
ALTER TABLE prescription ADD TRANSACTION;

CHANGE SYSDATE TO '1993/04/05';
DELETE FROM prescription
WHERE Patient_Id = 1734654 AND Drug_Id = 16584;

CHANGE SYSDATE TO '1993/04/10';
INSERT INTO patient VALUES (
  8839782,
  'Frank Zappa',
  'Los Angeles',
  '1940/12/21' );
COMMIT;

CHANGE SYSDATE TO '1994/01/03';
INSERT INTO patient VALUES (
  9365822,
  'Kurt Cobain',
  'Seattle',
  '1967/02/20' );
COMMIT;

/* Next we use TUC queries to query the current state of
   relations and views
*/
SELECT * FROM patient;
SELECT * FROM prescription;
SELECT * FROM addicted;
```

```
/* With TUC queries we can only access the current state; it is
   impossible to retrieve historical or future information.  To
   view the entire content of the database we employ two sequenced
   queries (see below).
*/
SEQUENCED VALID SELECT * FROM patient;
SEQUENCED VALID AND SEQUENCED TRANSACTION SELECT * FROM prescription;

INSERT INTO patient VALUES (
  5596544, 'Freddie Mercury', 'London', '1946/09/05' );
COMMIT;

INSERT INTO prescription VALUES (
  7356378, 45477, 'Dr Quincy', SYSDATE );
COMMIT;
```

With TUC we can smoothly migrate a nontemporal database into a nontemporal database. TUC ensures that legacy applications don't have to change a single line when existing tables are altered to become temporal.

However TUC also heavily limits the benefits we can get from a temporal database. To exploit temporal information we have to employ sequenced and nonsequenced statements. We first explain sequenced statements, which provide lots of built-in support for statements that are based on snapshot-reducibility.

```
CHANGE SYSDATE TO '1994/01/04';
SET VALID PERIOD '1994/01/05 - 1994/01/10'
  INSERT INTO prescription VALUES (
    9365822, 17575, 'Dr Hook', '1994/01/03' );
COMMIT;

CHANGE SYSDATE TO '1994/01/08';
SET VALID PERIOD '1994/01/13 - 1994/01/20'
  INSERT INTO prescription VALUES (
    9365822, 38799, 'Dr Quincy', '1994/01/08' );
COMMIT;

/* Dr Hook corrects an erroneous DB entry */
CHANGE SYSDATE TO '1994/01/12';
SEQUENCED VALID
  DELETE FROM prescription
  WHERE Patient_Id = 9365822
  AND Drug_Id = 17575;
SET VALID PERIOD '1994/1/5 - 1994/1/15'
```

```
  INSERT INTO prescription VALUES (
    9365822, 17575, 'Dr Hook', '1994/01/03' );
COMMIT;

CHANGE SYSDATE TO '1996/06/12';
SET VALID PERIOD '1984/6/26 - 1984/11/30'
  INSERT INTO patient VALUES (
    7565836, 'John Lennon', 'London', '1940/10/09' );
SET VALID PERIOD '1984/6/26 - 1984/11/30'
  INSERT INTO prescription VALUES (
    7565836, 38799, 'Dr Hook', '1984/7/11' );
SET VALID PERIOD '1996/6/20 - NOW'
  INSERT INTO patient VALUES (
    7565836, 'John Lennon', 'New York City', '1940/10/09' );
SET VALID PERIOD '1996/6/20 - NOW'
  INSERT INTO prescription VALUES (
    7565836, 69111, 'Dr Hook', '1996/6/12');
COMMIT;

SEQUENCED VALID AND SEQUENCED TRANSACTION PERIOD '1994/1/8'
  SELECT * FROM prescription;

SEQUENCED VALID
  UPDATE patient
  SET    Name = 'Jim Morrison'
  WHERE  Name = 'Jim Morison';
COMMIT;

SEQUENCED VALID
  SELECT p1.Patient_Id
  FROM prescription p1, prescription p2
  WHERE p1.Patient_Id = p2.Patient_Id
  AND   p1.Doctor <> p2.Doctor;

SEQUENCED VALID
  SELECT * FROM patient;
SEQUENCED VALID AND SEQUENCED TRANSACTION
  SELECT * FROM prescription;

SEQUENCED VALID
  SELECT proz.Doctor, VTIME(proz), morph.Doctor, VTIME(morph)
  FROM prescription proz, prescription morph
  WHERE proz.Drug_Id = 17575 /* Prozac */
  AND   proz.Patient_Id = 9365822 /* Kurt Cobain */
```

```
  AND   morph.Drug_Id = 38799 /* Morphine */
  AND   morph.Patient_Id = 9365822/* Kurt Cobain */;

SEQUENCED TRANSACTION AND SEQUENCED VALID
  SELECT *
  FROM prescription
  WHERE Patient_Id = 9365822/* Kurt Cobain */;

CREATE TABLE drug(Drug_Id INTEGER SEQUENCED VALID PRIMARY KEY,
                  Name VARCHAR(32),
                  Supplier VARCHAR(32) NOT NULL) AS VALID;

CREATE ASSERTION not_prozac_and_morphine CHECK
  SEQUENCED VALID PERIOD '1996/7/1 - NOW'
    ( NOT EXISTS ( SELECT *
                   FROM  prescription proz, prescription morph
                   WHERE proz.Drug_Id = 17575 /* Prozac */
                   AND   morph.Drug_Id = 38799 /* Morphine */
                   AND   morph.Patient_Id = proz.Patient_Id ));

CHANGE SYSDATE TO '1996/08/23';
( SEQUENCED VALID
    SELECT Patient_Id, Name
    FROM prescription, patient
    WHERE prescription.Patient_Id = patient.Id
)(VALID);
```

Finally, nonsequenced statements account for temporal statements that require the specification of special temporal relationships.

```
NONSEQUENCED VALID
  SELECT p1.Patient_Id
  FROM prescription p1, prescription p2
  WHERE p1.Patient_Id = p2.Patient_Id
  AND   p1.Doctor <> p2.Doctor
  AND   VTIME(p1) PRECEDES VTIME(p2);

CHANGE SYSDATE TO '1996/10/15';

NONSEQUENCED VALID
  SELECT Patient_Id, Name, DURATION(VTIME(some_prescription), MONTH)
  FROM ( SEQUENCED VALID
           SELECT patient.Patient_Id, Name
           FROM prescription, patient
```

```
          WHERE prescription.Patient_Id = patient.Patient_Id
      )(VALID) AS some_prescription
  WHERE DURATION(VTIME(some_prescription), MONTH) > 20;

DROP ASSERTION not_prozac_and_morphine;
DROP VIEW addicted;
DROP TABLE drug;
DROP TABLE prescription;
DROP TABLE patient;
```

Chapter 7

Test Files

This chapter consists of a set of files that are used to test Tiger. Browsing through these files gives you an idea of the (current) functionality of Tiger.

7.1 Migrating Databases

```
create table p(a integer)
*** Executing an UC statement ***.
statement processed

insert into p values(6)
....*** Executing an UC statement ***.
1 tuple processed

insert into p values(7)
....*** Executing an UC statement ***.
1 tuple processed

alter table p add valid
......
Table altered

drop table p
.......
Table dropped

create table p1(a integer primary key)
*** Executing an UC statement ***.
statement processed
```

```
create table q1(b integer references p1(a))
.*** Executing an UC statement ***.
statement processed

create table r1(b integer references p1)
.*** Executing an UC statement ***.
statement processed

insert into p1 values(7)
....*** Executing an UC statement ***.
1 tuple processed

insert into q1 values(7)
....*** Executing an UC statement ***.
1 tuple processed

commit
.
Transaction commited

insert into r1 values (8)
....*** Executing an UC statement ***.
check
.
No integrity constraints violated

commit
.
Transaction commited

alter table p1 add valid
..................
Table altered

drop table q1
......*** Executing an UC statement ***.
statement processed

drop table r1
......*** Executing an UC statement ***.
statement processed

drop table p1
```

```
.......
Table dropped

create table p(a integer)
*** Executing an UC statement ***.
statement processed

insert into p values(8)
....*** Executing an UC statement ***.
1 tuple processed

insert into p values(7)
....*** Executing an UC statement ***.
1 tuple processed

insert into p values(5)
....*** Executing an UC statement ***.
1 tuple processed

create view v as select * from p where a>6
.*** Executing an UC statement ***.
statement processed

select * from v
.*** Executing an UC statement ***.

A
---------------------------------------
8
7

alter table p add valid
............
Table altered

select * from v
.*** Executing an UC statement ***.

A
---------------------------------------
8
7

delete from p where a =7
```

```
......
1 tuples deleted

select * from v
.*** Executing an UC statement ***.

A
----------------------------------------
8

drop view v
......*** Executing an UC statement ***.
statement processed

drop table p
.......
Table dropped
```

7.2 Data Manipulation Statements

```
ALTER SESSION SET NLS_DATE_FORMAT = 'YYYY/MM/DD'
*** Executing an UC statement ***.
statement processed

create table p(a integer) as valid
.
Table created

insert into p values(2)
.....
1 tuples inserted

insert into p values(4)
.....
1 tuples inserted

insert into p values(8)
.....
1 tuples inserted

delete from p where a=4
.....
```

```
1 tuples deleted

nonsequenced valid delete from p where a=4
.....
1 tuples deleted

nonsequenced valid period '9-22' delete from p where a=4
.drop table p
.......
Table dropped

create table pbi(a integer) as valid and transaction
.
Table created

delete from pbi where a=4
......
0 tuples deleted

drop table pbi
.......
Table dropped
```

Chapter 8

Conclusions and Future Plans

I hope you enjoy working with Tiger and I also hope that it helps you to get an understanding of crucial concepts of ATSQL.

Chapter 9

Acknowledgements

I'd like to thank Renato Busatto and Christina Zennaro who contributed to the development of Tiger. Renato worked on various aspects of Tiger and he also contributed to the formalization of the semantics of ATSQL. Christina developed a first version of the world wide web interface.

My very special thanks to Robert Marti who was my PhD advisor at ETH Zrich when I started working on ChronoLog, a temporal deductive database system that provided many valuable input to Tiger and ATSQL. Special thanks also to Hans-Jrg Schek for co-advising this work. Next, I'm indebted to Rick Snodgrass who hosted me as a postdoc at the University of Arizona. He let me take part in his work and together with him I started to work on the language design of ATSQL. My very special thanks to Christian Jensen. We closely collaborate in the database group at Aalborg University where we completed the design of ATSQL.

Bibliography

[BJ96] M. H. Böhlen and C. S. Jensen. A Seamless Integration of Time into SQL. *submitted to ACM Transactions on Database Systems*, December 1996.

[BSS96] M. H. Böhlen, R. T. Snodgrass, and M. D. Soo. Coalescing in Temporal Databases. In T. M. Vijayaraman, A. Buchmann, C. Mohan, and N. L. Sarda, editors, *Proceedings of the Twenty-second International Conference on Very Large Data Bases*, pages 180–191. Morgan Kaufmann Publishers, Inc., Mumbai (Bombay), India, September 1996.

[Bur92] J. Burse. ProQuel: Using Prolog to Implement a Deductive Database System. Technical Report 191, Departement für Informatik, ETH Zürich, Switzerland, December 1992.

[CM87] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer Verlag, Berlin, 3rd edition, 1987.

[GT91] A. Van Gelder and R. W. Topor. Safety and Translation of Relational Calculus Queries. *ACM Transactions on Database Systems*, 16(2):235–278, June 1991.

[MS93] J. Melton and A. R. Simon. *Understanding the new SQL: A Complete Guide.* Morgan Kaufmann Publishers, San Mateo, California, 1993.

[Nev86] D. Neville. *ORACLE Pro*C User's Guide.* Oracle Corporation, Belmont, California, USA, 1.0 edition, 1986.

[SA85] R. T. Snodgrass and I. Ahn. A Taxonomy of Time in Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1985.

[Sno95] R. T. Snodgrass. *The TSQL2 Temporal Query Language.* Kluwer Academic Publishers, Boston, 1995.

[Wir86] N. Wirth. *Compilerbau.* Teubner Sudienbücher, Informatik, B. G. Teubner, Stuttgart, 1986.