

IBS

INSTITUTE OF BEHAVIORAL SCIENCE ■

POPULATION AGING CENTER ■

University of Colorado at Boulder
Boulder CO 80309-0483

WORKING PAPER PAC2003-0002 ■

An Introduction to the Relational Model of Data and the Structured Query Language for Demographers

Samuel Clark

February 14, 2003

An Introduction to the Relational Model of Data and the Structured Query Language for Demographers

Samuel Clark

Abstract

The Relational Model of Data (RMD) underlies the vast majority of commercially available database management systems sold today, and the Structured Query Language (SQL) is the language through which *relational databases* are accessed and manipulated. It is in the nature of Demography to store and manipulate large quantities of information, and consequently, it is useful for Demographers to be conversant with the RMD and SQL.

This communication provides an ultra brief introduction to the RMD and SQL, assuming the reader has no background in the theory of data management. Following that, a number of SQL queries that are particularly relevant to Demography are presented and discussed, including counts and frequencies, the calculation of person years, event/exposure rates, and the creation of person-time datasets suitable for event history analysis.

Because this communication aims to impart some practical knowledge, all of the demographically relevant queries are constructed in a Microsoft Access database that is freely available for download as part of this communication.

Version: 0

Date: 2003-02-14

Table of Contents

1	The Relational Model of Data	3
1.1	Relations	3
1.2	Operators	3
1.3	Constraints	4
2	The Structured Query Language	5
2.1	DDL - Data Definition Language	5
2.1.1	CREATE TABLE Example	5
2.1.2	ALTER TABLE Example	6
2.1.3	DROP TABLE Example	6
2.2	DML - Data Manipulation Language	6
2.2.1	INSERT Example	6
2.2.2	UPDATE Example	7
2.2.3	DELETE Example	8
2.2.4	SELECT and UNION Examples	8
2.2.5	ORDER BY and GROUP BY Clauses	11
2.3	SQL in Practical Use	15
3	Examples of SQL Queries for Demographers	17
3.1	A Useful <i>Desktop</i> Relational Database Microsoft Access	17
3.1.1	MS Access JOIN Syntax	18
3.1.2	MS Access User-Defined Functions	19
3.2	Tables & Queries Used in Examples	19
3.2.1	People Table	19
3.2.2	Ages Table	20
	Sexes Table	21
3.2.3	Sex-Age-Groups Query	21
3.2.4	Dates Table	23
3.2.5	SQL Queries for Demographers	23
4	Counts and Frequencies	24
4.1	Explanation of query: qrySelectFrequencyBySexAge_General	25
4.2	Explanation: qrySelectFrequencyBySexAge_General_AllAges	26
4.3	A Further Note on Counts and SQL	27
5	Person-Years	28
5.1	Explanation of function: fnIntervalIntersection	34
5.2	Explanation of query: qrySelectPersonYearsBySexAge_General	34
6	Event-Exposure Rates	37
6.1	Explanation of query: qrySelectMortalityRate_General	39
7	Person-Time Format Tables	40
7.1	Explanation of query: qryMakePersonYears	42
8	Summary	44
9	Bibliography	45

1 The Relational Model of Data

The relational model of data is a theory describing a consistent framework in which to structure, manipulate and maintain the consistency of data (Codd 1970, 1990). Data are stored and manipulated in relations (or tables), a set of operators manipulates relations to produce other relations, and relations are subject to certain constraints that insure the integrity of the data. For a lucid discussion of the relational model and relational database systems, refer to [An Introduction to Database Systems](#) by C.J. Date (Date 2000).

1.1 Relations

A relation is composed of a heading and a body. The heading is a *predicate* composed of a number of attributes, each with its own domain of possible values. The body is a collection of *tuples* (or rows) that represent true propositions formed by choosing a value from the domain of each of the attributes specified in the predicate. Each tuple represents a set of attribute values that when substituted into the predicate form a true proposition. As a consequence:

- There is no order specified (or necessary) for the attributes;
- There is no order specified (or necessary) for the tuples;
- Each tuple contains a unique value for each attribute; and,
- There are no duplicate tuples (each true proposition is recorded exactly once).

1.2 Operators

The operators used to manipulate relations are similar, and in large part analogous, to the familiar set operators in mathematics. In fact, relations are carefully defined sets. All of the operators take relations as arguments and produce a relation as their result – this is the *relational closure property*. Closure is critically important because it insures that one can write and evaluate nested relational expressions; those in which an expression can be substituted in place of a relation. The most fundamental operators used to manipulate relations include:

- Restrict;
- Project;
- Union; and,
- Join.

Restrict produces a relation whose tuples satisfy a condition. In other words, Restrict allows one to select a subset of the tuples in a relation. Project produces a relation with only the attributes specified in the Project expression. The result contains all of the tuples of the original relation with only those attributes specified in the Project expression. Restrict produces a row-wise subset, and Project produces an attribute-wise subset.

Union produces a relation containing all of the tuples from two specified relations of the same type. Because all tuples in a relation must conform to the

same predicate, the two relations contributing to the union must be based on the same predicate, or in other words, be of the same type. This insures that the result is a valid relation, and closure is maintained.

Join produces a relation that consists of all tuples produced by forming all possible pairs of tuples from two specified relations. Most often, there is at least one common attribute shared by the two relations, and the result of the join is subject to the condition that the values of the common attribute in both relations are equal. Join does precisely what its name implies - it takes two relations and 'glues' them together into a 'wider' relation. If you imagine the input relations as sets of tuples, Join takes the Cartesian product of the two sets of tuples, and then selects only those new tuples for which the values of the common attribute(s) of the input tuples are equal. There are many different varieties of the Join operator, the one described here being the most general.

In addition to the manipulation operators, there is a set of operators used to create and modify relations and update the values of their attributes. These will be discussed briefly in the section 2.1 below.

1.3 Constraints

The two crucial data integrity constraints have to do with unique values that identify tuples. A *primary key* is an attribute whose value is unique across all tuples in a relation. In general it is useful for all relations to have a primary key. The primary key can consist of a calculated attribute that combines values from other attributes to yield a unique value. However, for reasons not discussed here, it is often cleaner and more efficient to have a single-attribute unique key in each relation.

Tuples in two relations often share a semantic relationship that is represented by a link between the relations. Depending on the cardinality of the relationship (how many tuples in each relation are related to how many tuples in the other), primary key values from the parent relation are stored in a special attribute of the child relation. This attribute is called a *foreign key*. For each tuple in the child relation the foreign key attribute contains the (unique) value of the primary key attribute of a tuple in the parent relation. Foreign keys are constrained in the sense that a value stored in a foreign key attribute must exist as a value of the primary key in one of the tuples of the parent relation.

In circumstances where tuples in the child relation are nonsensical in the absence of corresponding tuples in the parent relation, the use of primary and foreign key constraints insures that no such orphan (and meaningless) tuples ever exist in the child relation.

Constraints come in *many* other forms including the careful construction of the domains (or types) from which attribute values may be drawn and manipulated.

•

2 The Structured Query Language

SQL or Structured (or Standard) Query Language was invented and first implemented by IBM in the 1970s as SEQUEL (Structured English Query Language). SQL is the practical specification of the relational model, and working implementations of SQL are offered by many database systems vendors. Most of the widely available implementations of SQL are based on the International Standard Database Language SQL (1992) or SQL/92. Recently, an updated version of the specification has been produced and is referred to as SQL/99. However, at the time of this writing SQL/99 has not yet been well implemented by any database systems vendor.

In very broad terms, SQL has three components:

- Data Definition Language or **DDL** components;
- Data Manipulation Language or **DML** components; and,
- Management components.

2.1 DDL - Data Definition Language

The data definition statements allow the user to create, modify and drop (or delete) tables (what we discussed above as 'relations') and statements to manage various other aspects of the data architecture of the database. For our purposes the important statements are:

- CREATE TABLE - create and define a new table;
- ALTER TABLE - modify the definition of a table; and,
- DROP TABLE - delete a table from the database.

Following are examples of these commands and the output they produce using IBM's implementation of SQL, available in their DB2 Universal Database management system. Three tables are created and will be used in the next set of examples.

2.1.1 CREATE TABLE Example

Example 2-1 creates a table named 'people' with fields (attributes) id, name, and sex. id is a primary key, and none of the fields may contain a null value. For each field, the data type is defined after the field name and is self-explanatory.

```
CREATE TABLE people
  (id Integer NOT NULL PRIMARY KEY,
   name Varchar(20) NOT NULL,
   sex Char(1) NOT NULL);
```

Example 2-1

Example 2-2 creates a table named 'locations' with fields id, name, and type. Again, id is the primary key, and none of the fields can contain a null value.

```
CREATE TABLE locations
  (id Integer NOT NULL PRIMARY KEY,
   name VarChar(20) NOT NULL,
   Type Char(1) NOT NULL);
```

Example 2-2

Example 2-3 creates a table named 'lives_at' with fields id, pid, and lid. This is

a many-to-many *relationship* table that stores information linking people to the places where they live; hence, the name 'lives_at'. The id field is a primary key, no nulls are allowed in any fields, and the two fields pid and lid store values from the id fields of the people and locations tables and are, therefore, foreign keys.

```
CREATE TABLE lives_at
  (id Integer NOT NULL PRIMARY Key,
  pid Integer NOT NULL,
  lid Integer NOT NULL,
  FOREIGN KEY (pid) REFERENCES people (id),
  FOREIGN KEY (lid) REFERENCES locations (id));
```

Example 2-3

2.1.2 ALTER TABLE Example

Example 2-4 adds a date of birth (dob) field of type 'Date' to the people table.

```
ALTER TABLE people
  ADD COLUMN dob Date;
```

Example 2-4

2.1.3 DROP TABLE Example

Example 2-5 deletes the people table from the database.

```
DROP TABLE people;
```

Example 2-5

2.2 DML - Data Manipulation Language

The data manipulation statements allow the user to manipulate and modify the data stored in the tables. These include:

- INSERT;
- UPDATE;
- DELETE;
- SELECT: GROUP BY & ORDER BY;
- UNION;
- INTERSECT, DIVIDE, EXCLUDE and many less-used operators.

We will discuss INSERT through UNION and leave the interested reader to investigate the others.

2.2.1 INSERT Example

Example 2-6 populates the people table with the values listed below:

```

INSERT INTO people (id,name,sex)
VALUES (1,'Mary','F'),
       (2,'Paul','M'),
       (3,'Simon','M'),
       (4,'Jane','F'),
       (5,'Albert','M'),
       (6,'Ruth','F');

```

Example 2-6

ID	NAME	SEX	DOB
1	Mary	F	-
2	Paul	M	-
3	Simon	M	-
4	Jane	F	-
5	Albert	M	-
6	Ruth	F	-

Example 2-7 populates the locations table with the values listed below:

```

INSERT INTO locations (id,name,type)
VALUES (1,'Nairobi','U'),
       (2,'Accra','U'),
       (3,'Mtubatuba','R'),
       (4,'Sinafala','R'),
       (5,'Cairo','U');

```

Example 2-7

ID	NAME	TYPE
1	Nairobi	U
2	Accra	U
3	Mtubatuba	R
4	Sinafala	R
5	Cairo	U

Example 2-8 populates the lives_at table with the listed values, shown below:

```

INSERT INTO lives_at (id,pid,lid)
VALUES (1,1,4),
       (2,2,5),
       (3,3,2),
       (4,4,2),
       (5,5,1),
       (6,6,3);

```

Example 2-8

ID	PID	LID
1	1	4
2	2	5
3	3	2
4	4	2
5	5	1
6	6	3

2.2.2 UPDATE Example

Example 2-9 updates the (new) dob field in the people table with 06/04/1950 for Mary whose id is 1 , shown below after an update of all six records (tuples) in the people table:

```

UPDATE people
  SET dob = '1950-6-4'
 WHERE id = 1;

UPDATE people
  SET dob = '1966-10-12'
 WHERE id = 2;

UPDATE people
  SET dob = '1980-4-1'
 WHERE id = 3;

UPDATE people
  SET dob = '1981-9-12'
 WHERE id = 4;

UPDATE people
  SET dob = '1991-12-2'
 WHERE id = 5;

UPDATE people
  SET dob = '1991-7-11'
 WHERE id = 6;

```

Example 2-9

ID	NAME	SEX	DOB
1	Mary	F	06/04/1950
2	Paul	M	10/12/1966
3	Simon	M	04/01/1980
4	Jane	F	09/12/1981
5	Albert	M	12/02/1991
6	Ruth	F	07/11/1991

2.2.3 DELETE Example

Example 2-10 deletes the record(s) in table `lives_at` for which `id` is equal to 8.

```

DELETE FROM lives_at
 WHERE id = 8;

```

Example 2-10

2.2.4 SELECT and UNION Examples

Example 2-11 produces the Cartesian product of the `people`, `lives_at` and `locations` tables and then selects the resulting records for which the `id` field from the `people` table is equal to the `pid` field from the `lives_at` table and the `lid` field from the `lives_at` table is equal to the `id` field from the `locations` table. Once the records fulfilling those conditions are identified, the `name` fields from the `people` and `locations` tables are selected, and the records are listed. This effectively *joins* the `people`, `lives_at`, and `locations` tables *on* the field pairs (`people.id`, `pid`) and (`lid`, `locations.id`) subject to the condition that the resulting records must have equal values for those pairs of fields. In terms of the relational model, this select statement performs a join and a projection. The join is accomplished through the Cartesian product and the conditions placed on the `id` fields. This statement makes the join explicit and equivalent to a Cartesian product followed by a restriction; there is a more compact syntax that accomplishes the same thing without specifying the equalities explicitly, presented below in 3.1 below. The projection is accomplished through the selection of the `name` fields for retrieval from the result table. The `order by` clause sorts the resulting table on

the name field from the people table. The statement is run on the tables created in the previous examples, and the result is shown below:

```
SELECT p.name AS name, l.name AS location
FROM people AS p, lives_at AS la, locations AS l
WHERE p.id = la.pid
      AND la.lid = l.id
ORDER BY p.name;
```

NAME	LOCATION
Albert	Nairobi
Jane	Accra
Mary	Sinafala
Paul	Cairo
Ruth	Mtubatuba
Simon	Accra

Example 2-11

Example 2-12 is the same as Example 2-11 with the addition of an additional restriction, `p.dob < '1980-01-01'`, that retrieves only records for which the date of birth is prior to January 1, 1980, below:

```
SELECT p.name AS name, l.name AS location
FROM people AS p, lives_at AS la, locations AS l
WHERE p.id = la.pid
      AND la.lid = l.id
      AND p.dob < '1980-01-01'
ORDER BY p.name;
```

NAME	LOCATION
Mary	Sinafala
Paul	Cairo

Example 2-12

Example 2-13 produces a full Cartesian product of the people table with itself. In this case no restriction is put on the result of the Cartesian product resulting in a table consisting of records that pair each record in the people table with every other record in the people table, including itself. Again, a projection is performed to retrieve just the name fields from the resulting table. Last, the order by clause at the end of this statement sorts the resulting table first on the name field from the first instance of the people table and then on the name field of the second instance of the people table. The final result is shown below in Example 2-13.

```

SELECT p1.name AS name1, p2.name AS name2
FROM people AS p1, people AS p2
ORDER BY p1.name, p2.name;

```

NB: No 'WHERE' clause resulting in the complete Cartesian product of the people table with itself

NAME1	NAME2
-----	-----
Albert	Albert
Albert	Jane
Albert	Mary
Albert	Paul
Albert	Ruth
Albert	Simon
Jane	Albert
Jane	Jane
Jane	Mary
Jane	Paul
Jane	Ruth
Jane	Simon
Mary	Albert
Mary	Jane
Mary	Mary
Mary	Paul
Mary	Ruth
Mary	Simon
Paul	Albert
Paul	Jane
Paul	Mary
Paul	Paul
Paul	Ruth
Paul	Simon
Ruth	Albert
Ruth	Jane
Ruth	Mary
Ruth	Paul
Ruth	Ruth
Ruth	Simon
Simon	Albert
Simon	Jane
Simon	Mary
Simon	Paul
Simon	Ruth
Simon	Simon

Example 2-13

Example 2-14 selects all of the records from the people table and performs a union of those records to create the result below in which all of the records in the people table appear twice (in violation of one of the basic precepts of the Relational Model of Data because the keyword 'ALL' was used after UNION – without the 'ALL' option identical records only appear once in the table resulting from a UNION command). The second version of this query demonstrates a nested relational expression (the UNION expression). The UNION is performed first and then the outer SELECT statement sorts the result by name, below:

```

SELECT *
FROM people
UNION ALL
SELECT *
FROM people;

```

ID	NAME	SEX	DOB
1	Mary	F	06/04/1950
2	Paul	M	10/12/1966
3	Simon	M	04/01/1980
4	Jane	F	09/12/1981
5	Albert	M	12/02/1991
6	Ruth	F	07/11/1991
1	Mary	F	06/04/1950
2	Paul	M	10/12/1966
3	Simon	M	04/01/1980
4	Jane	F	09/12/1981
5	Albert	M	12/02/1991
6	Ruth	F	07/11/1991

```

SELECT *
FROM
  (SELECT *
   FROM people
  UNION ALL
   SELECT *
   FROM people) AS individuals
ORDER BY name;

```

Example 2-14

ID	NAME	SEX	DOB
5	Albert	M	12/02/1991
5	Albert	M	12/02/1991
4	Jane	F	09/12/1981
4	Jane	F	09/12/1981
1	Mary	F	06/04/1950
1	Mary	F	06/04/1950
2	Paul	M	10/12/1966
2	Paul	M	10/12/1966
6	Ruth	F	07/11/1991
6	Ruth	F	07/11/1991
3	Simon	M	04/01/1980
3	Simon	M	04/01/1980

2.2.5 ORDER BY and GROUP BY Clauses

The ORDER BY clause of the SELECT statement simple sorts the resulting table by one or more fields in either ascending or descending order; see Example 2-11 through Example 2-13, above.

The GROUP BY clause provides a powerful way to subset the records resulting from a SELECT statement and to perform calculations on each subset. The argument of the GROUP BY clause is one or more fields, and the result is a table with sets of records; each set having identical values of the fields specified as arguments of GROUP BY. The functions used to calculate over these groups are called 'domain aggregate' functions because they calculate a value over an aggregate of one of the *domains* (the proper name of a field). Example 2-15, below demonstrates the use of both the GROUP BY and ORDER BY clauses.

New people table after adding Louise, Adam, Bill and Madeline:

ID	NAME	SEX	DOB
1	Mary	F	06/04/1950
2	Paul	M	10/12/1966
3	Simon	M	04/01/1980
4	Jane	F	09/12/1981
5	Albert	M	12/02/1991
6	Ruth	F	07/11/1991
7	Louise	F	03/14/1986
8	Adam	M	11/01/1964
9	Bill	M	03/10/1997
10	Madeline	F	03/26/2001

New lives_at table after adding lives_at links for the four new people:

ID	PID	LID
1	1	4
2	2	5
3	3	2
4	4	2
5	5	1
6	6	3
7	7	4
8	8	5
9	9	2
10	10	2

Example 2-15

NB: The locations table remains unchanged

```
SELECT l.name AS location, p.id AS personid
FROM people AS p, lives_at AS la, locations AS l
WHERE p.id = la.pid
AND la.lid = l.id
ORDER BY l.name DESC;
```

LOCATION	PERSONID
Sinafala	1
Sinafala	7
Nairobi	5
Mtubatuba	6
Cairo	2
Cairo	8
Accra	3
Accra	4
Accra	9
Accra	10

```

SELECT l.name AS location, count(p.id) AS frequency
FROM people AS p, lives_at AS la, locations AS l
WHERE p.id = la.pid
      AND la.lid = l.id
GROUP BY l.name
ORDER BY l.name DESC;

```

LOCATION	FREQUENCY
Sinafala	2
Nairobi	1
Mtubatuba	1
Cairo	2
Accra	4

```

SELECT l.name AS location, max(p.id) AS MaximumValue
FROM people AS p, lives_at AS la, locations AS l
WHERE p.id = la.pid
      AND la.lid = l.id
GROUP BY l.name
ORDER BY l.name;

```

LOCATION	MAXIMUMVALUE
Accra	10
Cairo	8
Mtubatuba	6
Nairobi	5
Sinafala	7

Four new people are added to the people table with ID numbers from 7 to 10 in order to have more than one location with multiple residents. The first example query identifies the ID numbers of all of the people living at each location, and it is obvious that several locations have more than one resident. The next two queries group over location and calculate the *count* and the *maximum* (max) of the people.id field in each group identified by a location.

Example 2-16 below provides a more useful look at the GROUP BY and ORDER BY clauses. Each person's age in years is calculated on October 22, 2002 and displayed rounded to two decimal places using this bit of code:

```
AGE = DECIMAL(ROUND((DAYS('2002-10-22')-DAYS(p.dob))/365.25,2),4,2)
```

The first query using AGE groups over sex and calculates the average age for each sex. The next produces a table with one row for each person listing their name, location and age, and this forms the basis for the next two queries. The final two queries group by location and calculate the number of people at each location and the sum and averages of their ages, respectively.

```

SELECT p.name, p.sex, DECIMAL(ROUND((DAYS('2002-10-22')-
  DAYS(p.dob))/365.25,2),4,2) AS age20021022
FROM people AS p;

```

NAME	SEX	AGE20021022
Mary	F	52.38
Paul	M	36.03
Simon	M	22.56
Jane	F	21.11
Albert	M	10.89
Ruth	F	11.28
Louise	F	16.61
Adam	M	37.97
Bill	M	5.62
Madeline	F	1.57

```

SELECT p.sex, DECIMAL(ROUND(avg((DAYS('2002-10-22')-
  DAYS(p.dob))/365.25),2),4,2) AS avg_age20021022
FROM people AS p
GROUP BY p.sex
ORDER BY p.sex;

```

SEX	AVG_AGE20021022
F	20.59
M	22.61

Example 2-16

```

SELECT p.name AS name, l.name AS location, DECIMAL(ROUND((DAYS('2002-10-
  22')-DAYS(p.dob))/365.25,2),4,2) AS age20021022
FROM people AS p, lives_at AS la, locations AS l
WHERE p.id = la.pid
  AND la.lid = l.id
ORDER BY l.name, p.name;

```

NAME	LOCATION	AGE20021022
Bill	Accra	5.62
Jane	Accra	21.11
Madeline	Accra	1.57
Simon	Accra	22.56
Adam	Cairo	37.97
Paul	Cairo	36.03
Ruth	Mtubatuba	11.28
Albert	Nairobi	10.89
Louise	Sinafala	16.61
Mary	Sinafala	52.38

```

SELECT l.name AS location, count(p.id) AS frequency,
       DECIMAL(ROUND(sum((DAYS('2002-10-22')-DAYS(p.dob))/365.25),2),4,2) AS
       sum_age20021022
FROM people AS p, lives_at AS la, locations AS l
WHERE p.id = la.pid
      AND la.lid = l.id
GROUP BY l.name
ORDER BY l.name;

```

LOCATION	FREQUENCY	SUM_AGE20021022
Accra	4	50.86
Cairo	2	74.00
Mtubatuba	1	11.28
Nairobi	1	10.89
Sinafala	2	68.99

```

SELECT l.name AS location, count(p.id) AS frequency,
       DECIMAL(ROUND(avg((DAYS('2002-10-22')-DAYS(p.dob))/365.25),2),4,2) AS
       avg_age20021022
FROM people AS p, lives_at AS la, locations AS l
WHERE p.id = la.pid
      AND la.lid = l.id
GROUP BY l.name
ORDER BY l.name;

```

LOCATION	FREQUENCY	AVG_AGE20021022
Accra	4	12.71
Cairo	2	37.00
Mtubatuba	1	11.28
Nairobi	1	10.89
Sinafala	2	34.50

2.3 SQL in Practical Use

In practice, SQL is used in three ways. SQL can be used in a dynamic sense in either of two ways: 1) a user can type SQL into a command line interpreter (CLP) that parses the statements and passes them to the database management system for execution in real time, or 2) a program can construct SQL in real time based on parameter values or input from a user and then pass the SQL to the interpreter and onward to the data base management system. In both of these situations, the SQL is interpreted and optimized in real time – hence the term 'dynamic' SQL. Static SQL consists of predefined, pre-optimized SQL statements that are usually invoked by a program and cannot be altered in real time. Static SQL usually runs faster because the interpretation, optimization and compile steps need not be performed, and for this reason, operations that are run many times without change are best implemented as static SQL.

Although SQL is a computational complete language, it is usually used as a data manipulation sublanguage, that in combination with a fully-featured host language, provides all of the functionality necessary to manage and manipulate the data, to perform the logic necessary for the application to run, and to interact with the user. Common host languages include C, C++, Java, Visual Basic and Cobol.

-

3 Examples of SQL Queries for Demographers

3.1 A Useful *Desktop* Relational Database Management System: Microsoft Access

All of the example calculations are demonstrated using concrete examples developed in a Microsoft Access XP database named "SQLForDemographers.mdb" which is included as a downloadable appendix to this communication. Microsoft Access is ideal for single-user, desktop use. Although Access does not support the full range of SQL capability, it does support all of the core SQL capabilities, it provides a clean user-friendly graphical interface that is helpful for beginners, it is well-integrated with the other Microsoft Office applications, it stores all components of the database (including data) in one easily copied and moved file, and because it does extensive memory caching, it performs very well on small to medium-sized data sets. It is therefore recommended as a powerful desktop data manipulation tool appropriate for both beginner and advanced user. However, *a note of caution*: do not be tempted to use Access in multi-user, security-sensitive, large-scale database applications¹. Access lacks many of the advanced features necessary to implement large-scale relational databases, and it is far wiser to use Microsoft SQL Server, Oracle, IBM DB2 or some other *enterprise level* database management system for large, secure, reliability-sensitive relational database applications.

Before beginning, a brief discussion of some of Access's features is necessary. Access stores an entire database in a single file whose name is postfixed with ".mdb". After opening the file, Access presents the user with a selection of object types: TABLES, QUERIES, FORMS, REPORTS, PAGES, MACROS and MODULES. For our purposes, only the TABLES, QUERIES and MODULES objects are of interest. The data are stored in tables contained in the TABLES objects. SQL that operates on the tables is stored in QUERY objects, and Visual Basic code comprising user-defined functions is stored in MODULE objects.

Access has a powerful and easy to use query-by-example (QBE) interface for writing SQL; the impact of which is to make writing SQL in Access a largely *visual* exercise. This is both a plus and a minus; it allows novice users to get going quickly, but it masks (and makes inaccessible) much of the detail and richness of SQL. Right-clicking on a query in the QUERIES category and selecting 'Design View' opens up the graphical query writing tool, and it is left to the reader to explore its functionality. However, it is worth noting that a right-click on the open query design view provides an option to move to 'SQL View'. Selecting that option opens up a text view in which you can edit or write the

¹ Access has many reliability problems in a multi-user environment, and additionally, some standard SQL queries produce wrong results when executed in Access. Additionally, some data consumers have policies that discourage consumption of data managed using Access. When reliability and accuracy are important, Access should be used with great care, or not at all. Microsoft's SQL Server and IBM's DB2 Database Management Systems are good alternatives. Access is discussed and used here *purely for convenience* and because many readers have access to Access and not to other more robust Relational Database Management Systems.

SQL that actually comprises the query, thus opening up some of the richer functionality of Access's SQL.

Finally, we must examine two Access-specific features of SQL before proceeding. The first is Access's syntax for the SQL JOIN operator, and the second is the way in which Access introduces user-defined functions that can be integrated into SQL statements.

3.1.1 MS Access JOIN Syntax

JOIN is a refinement of the syntax controlling the Cartesian product formed by two tables contributing to a query. The syntax for Access's JOIN is like this:

<table on left> [**LEFT, RIGHT or INNER**] **JOIN** <table on right> **ON** <conditions>

The <conditions> specify the relationship(s) between fields of the left and right tables that must evaluate to true in order for a result record to contain values from **both** records of the left and right tables. The LEFT, RIGHT and INNER options specify whether field values from records on the LEFT or RIGHT are included in the result when there is no matching record in the other table. For instance, the result of a LEFT JOIN includes a record corresponding to (and containing field values of) each record of the table on the left, but only contains values in the fields from the table on the right from matching records of the table on the right. Likewise, the result of a RIGHT JOIN is the same except that it contains a record corresponding to each record of the table on the right. An INNER JOIN contains only values from matching records of both tables; the result of an INNER JOIN contains one record for each set of matching records from the left and right tables. The behavior of Access's JOIN is demonstrated below in Example 3-1.

LEFT

ID	Name
1	Bill
2	Mary
3	Rachel

RIGHT

ID	Name
1	Rob
2	Mary
3	Tim

Example 3-1

```
SELECT tbILEFT.Name, tbIRIGHT.Name
FROM tbILEFT INNER JOIN tbIRIGHT ON tbILEFT.Name = tbIRIGHT.Name;
```

tbILEFT.Name	tbIRIGHT.Name
Mary	Mary

```
SELECT tbILEFT.Name, tbIRIGHT.Name
FROM tbILEFT LEFT JOIN tbIRIGHT ON tbILEFT.Name = tbIRIGHT.Name;
```

tbILEFT.Name	tbIRIGHT.Name
Bill	
Mary	Mary
Rachel	

```
SELECT tblLEFT.Name, tblRIGHT.Name
FROM tblLEFT RIGHT JOIN tblRIGHT ON tblLEFT.Name = tblRIGHT.Name;
```

tblLEFT.Name	tblRIGHT.Name
	Rob
Mary	Mary
	Tim

3.1.2 MS Access User-Defined Functions

In Microsoft Access, user-defined functions are written in Microsoft Visual Basic for applications, specifically the variety that is shipped with Microsoft office. They can be accessed through the MODULES category of the main control panel. Modules included with the example database are 'mdlAnalysis' which contains the interval intersection function, 'mdlPopulateDatesTable' which includes the subroutine to populate the dates table with an arbitrary collection of time intervals, and 'mdlPopulatePeopleTable' which contains the subroutine used to generate the Big People table.

Opening one of the modules in the Modules view starts the Visual Basic Editor and allows you to examine and edit the functions and subroutines. To invoke a function (a routine that returns a value) type "? <function name> (arg1, arg2, ...)" in the *Immediate Pane* below the editor pane (if it that is not visible, then use the View menu to make it visible). To invoke a subroutine (a routine that does not return a value), type <subroutine name> arg1, arg2, ... *without* the "?" or the "()".

3.2 Tables & Queries Used in Examples

3.2.1 People Table

All of the calculations operate on a table called **tblPeople** (Table 1, below) containing one row per individual and fields to record:

- The individual's ID: **ID**,
- The individual's sex: **Sex**,
- The individual's date of birth: **DOB**, and
- The individual's date of death: **DOD** (Null if alive at last observation).

Most of the examples presented here operate on the four people described in Table 1, the Small People table. There are only four people so that "by-hand" checking and verifying of the results can be done easily. A much larger people table, the Big People table, is included in the example database called 'tblPeopleBig' containing 1200 individuals created by the module called 'mdlPopulatePeopleTable'. It is left to the interested reader to understand how the larger people table is constructed and to investigate the operation of the example queries using the larger people table. A copy of the Small People table is also included called 'tblPeopleSmall'; to switch between Small and Big and back again, simply copy and paste the desired table giving it the name 'tblPeople'.

Table 1: tblPeople (Small People Table)

ID	Sex	DOB	DOD
1	Male	1/1/2000	12/31/2050
2	Female	1/1/2000	7/15/2000
3	Male	1/1/1980	12/31/2050
4	Female	1/1/2000	7/15/2080

3.2.2 Ages Table

In addition to the table containing the people whose demographic indicators are to be calculated, at least three other tables and one query are necessary to describe the attributes of those indicators:

- tblAges,
- tblSexes,
- tblDates, and
- qrySelectSexAgeGroups .

Each record in the Big Ages (Table 2, below) and Small Ages tables (Table 3, below) contains a description of an age group, and by editing, adding or deleting records in the ages table, one is easily able to customize the age groups used by the queries to calculate demographic indicators. The result is that the user does not need to edit SQL statements to alter the age groups applied in each calculation. The ages table contains fields to describe:

- Each age group's unique ID: **ID**,
- Each age group's label: **AgeGroup**,
- Each age group's starting age: **StartAge**, and
- Each age group's ending age: **EndAge**.

Again, to switch between Small and Big and back again, simply copy and paste the desired table giving it the name 'tblAges'.

Table 2: tblAges – Big Ages Table

ID	AgeGroup	StartAge	EndAge
1	0	0	0
2	1-4	1	4
3	5-9	5	9
4	10-14	10	14
5	15-19	15	19
6	20-24	20	24
7	25-29	25	29
8	30-34	30	34
9	35-39	35	39
10	40-44	40	44
11	45-49	45	49
12	50-54	50	54
13	55-59	55	59
14	60-64	60	64
15	65-69	65	69
16	70-74	70	74
17	75-79	75	79
18	80-84	80	84
19	85-89	85	89

ID	AgeGroup	StartAge	EndAge
20	90-94	90	94
21	95-99	95	99
22	100-104	100	104
23	105-109	105	109
24	110-114	110	114
25	115+	115	250

Table 3: tblAges – Small Ages Table

ID	AgeGroup	StartAge	EndAge
1	0	0	0
2	1-4	1	4
3	5-19	5	19
4	20-49	20	49
5	50+	50	250

Sexes Table

Like the ages table, the sexes table (Table 4, below) contains one row to describe each sex that needs to be taken into account in the calculations. Although “male” and “female” obviously dominate the list of sexes, it is often the case that many “sex unknown” values occur, and it is certainly the case that there are non-trivial numbers of people who are not “male”, or “female” and may be coded with some other label. So, in keeping with the metadata-centric approach to these calculations, the user is easily able to specify the sex categories that are necessary for the calculations s/he wants to make. The examples only use the customary male and female categories. The sexes table contains fields to describe:

- Each sex’s unique ID: **ID**,
- Each sex’s label: **Sex**,

Table 4: tblSexes

ID	Sex
1	Female
2	Male

3.2.3 Sex-Age-Groups Query

Finally, it is necessary to create a virtual table (the result of a query) that combines the sex and ages tables into a sex-age table that provides the sex-age categories that will be used to characterize the remainder of the calculations. This is accomplished by the SQL comprising the sex-age-groups query, Example 3-2. The sex-age-groups query performs a Cartesian product of the sex and ages tables to produce a new table containing one record *for each age group for each sex*.

```

SELECT tblSexes.Sex, tblAges.AgeGroup, tblAges.StartAge, tblAges.EndAge
FROM tblSexes, tblAges
ORDER BY tblSexes.Sex, tblAges.StartAge;

```

Example 3-2

NB: Notice that there is no WHERE clause in this SELECT statement; as a result there is no “restriction” on the Cartesian product formed by the two tables, and the whole Cartesian product is returned – each sex paired with each age.

Operating on the sexes and ages tables (Table 2), the sex-age-groups query produces the results displayed in Table 5, below.

Table 5: qrySelectSexAgeGroups

Sex	AgeGroup	StartAge	EndAge
Female	0	0	0
Female	1-4	1	4
Female	5-9	5	9
Female	10-14	10	14
Female	15-19	15	19
Female	20-24	20	24
Female	25-29	25	29
Female	30-34	30	34
Female	35-39	35	39
Female	40-44	40	44
Female	45-49	45	49
Female	50-54	50	54
Female	55-59	55	59
Female	60-64	60	64
Female	65-69	65	69
Female	70-74	70	74
Female	75-79	75	79
Female	80-84	80	84
Female	85-89	85	89
Female	90-94	90	94
Female	95-99	95	99
Female	100-104	100	104
Female	105-109	105	109
Female	110-114	110	114
Female	115+	115	250
Male	0	0	0
Male	1-4	1	4
Male	5-9	5	9
Male	10-14	10	14
Male	15-19	15	19
Male	20-24	20	24
Male	25-29	25	29
Male	30-34	30	34
Male	35-39	35	39
Male	40-44	40	44
Male	45-49	45	49
Male	50-54	50	54
Male	55-59	55	59
Male	60-64	60	64
Male	65-69	65	69
Male	70-74	70	74
Male	75-79	75	79
Male	80-84	80	84
Male	85-89	85	89
Male	90-94	90	94

Sex	AgeGroup	StartAge	EndAge
Male	95-99	95	99
Male	100-104	100	104
Male	105-109	105	109
Male	110-114	110	114
Male	115+	115	250

3.2.4 Dates Table

The dates table (Table 6, below) contains one record per time period to be included in a person-time data set. For example, this table contains one record for every year that the user wants to include in a Person-Year data set appropriate for event history analysis. The dates table contains fields to describe:

- Each time period's unique ID: **ID**,
- Each time period's begin date: **BeginDate**, and
- Each time period's end date: **EndDate**.

The example table contains dates describing calendar years from 1995 through 2109.

Table 6: tbIDates

ID	BeginDate	EndDate
1	1/1/1995	12/31/1995
2	1/1/1996	12/31/1996
3	1/1/1997	12/31/1997
4	1/1/1998	12/31/1998
5	1/1/1999	12/31/1999
6	1/1/2000	12/31/2000
:	:	:
110	1/1/2104	12/31/2104
111	1/1/2105	12/31/2105
112	1/1/2106	12/31/2106
113	1/1/2107	12/31/2107
114	1/1/2108	12/31/2108
115	1/1/2109	12/31/2109

3.2.5 SQL Queries for Demographers

The next four sections present the SQL form of several calculations commonly performed by Demographers:

- Counts and Frequencies,
- Person-Years,
- Event-Exposure Rates, and
- Creation of Person-Time Format Files.

•

4 Counts and Frequencies

The most basic form of demographic analysis simply involves counting – the number of people alive and resident somewhere at a given point in time; the number of male and female members of a population; the number of deaths occurring in a period of time; etc. This type of calculation usually done by a cross tabulation.

The GROUP BY clause and accompanying aggregate domain function *count()* of the SELECT statement make SQL naturally suited to this kind of calculation. Example 4-1, below displays the SQL for the All Ages Count query which calculates the sex -age-specific count of people alive on the date specified by the parameter [Date].

qrySelectFrequencyBySexAge_General_AllAges:

```
SELECT qrySelectSexAgeGroups.Sex, qrySelectSexAgeGroups.AgeGroup,
  IIf(IsNull(qrySelectFrequencyBySexAge_General.Frequency),0,qrySelectFrequen
  cyBySexAge_General.Frequency) AS Frequency
FROM qrySelectSexAgeGroups LEFT JOIN
  qrySelectFrequencyBySexAge_General ON
  (qrySelectSexAgeGroups.AgeGroup =
  qrySelectFrequencyBySexAge_General.AgeGroup) AND
  (qrySelectSexAgeGroups.Sex = qrySelectFrequencyBySexAge_General.Sex)
ORDER BY qrySelectSexAgeGroups.Sex, qrySelectSexAgeGroups.StartAge;
```

Example 4 -1

qrySelectFrequencyBySexAge_General:

```
PARAMETERS [Date] DateTime, [DaysInYear] IEEESingle;
SELECT tblPeople.Sex, tblAges.AgeGroup, Count(tblPeople.ID) AS Frequency
FROM tblPeople, tblAges
WHERE (((Int((1+[Date]-[DOB])/[DaysInYear]))>=[StartAge] And (Int((1+[Date]-
[DOB])/[DaysInYear]))<=[EndAge]) AND ((tblPeople.DOB)<=[Date]) AND
(((tblPeople.DOD)>=[Date]) Or ((tblPeople.DOD) Is Null)))
GROUP BY tblPeople.Sex, tblAges.AgeGroup, tblAges.StartAge
ORDER BY tblPeople.Sex, tblAges.StartAge;
```

In Example 4-1, the qrySelectFrequencyBySexAge_General_AllAges query calls the Count query which actually counts the number of people alive on [Date] by sex and age. The qrySelectFrequencyBySexAge_General_AllAges insures that all age groups for both sexes appear in the final result with a zero when the age group is empty. The results of running the query in Example 4-1 on the Big People table on January 1, 2050 are contained in Table 7, below:

**Table 7: qrySelectFrequencyBySexAge_General_AllAges
Run on the Big People Table on 1/1/2050**

Sex	AgeGroup	Frequency
Female	0	6
Female	1-4	20
Female	5-9	29
Female	10-14	31
Female	15-19	32
Female	20-24	32
Female	25-29	32
Female	30-34	32
Female	35-39	37

Sex	AgeGroup	Frequency
Female	40-44	33
Female	45-49	29
Female	50-54	0
Female	55-59	0
Female	60-64	0
Female	65-69	0
Female	70-74	0
Female	75-79	0
Female	80-84	0
Female	85-89	0
Female	90-94	0
Female	95-99	0
Female	100-104	0
Female	105-109	0
Female	110-114	0
Female	115+	0
Male	0	6
Male	1-4	28
Male	5-9	31
Male	10-14	29
Male	15-19	28
Male	20-24	28
Male	25-29	28
Male	30-34	28
Male	35-39	23
Male	40-44	27
Male	45-49	31
Male	50-54	1
Male	55-59	0
Male	60-64	0
Male	65-69	0
Male	70-74	0
Male	75-79	0
Male	80-84	0
Male	85-89	0
Male	90-94	0
Male	95-99	0
Male	100-104	0
Male	105-109	0
Male	110-114	0
Male	115+	0

4.1 Explanation of query: qrySelectFrequencyBySexAge_General

AIM: To count the number of people alive on [Date] by sex and age.

This query's name is postfixed with "_General" to indicate that its SQL does not need modification to produce tabulations differing by sex-age-groups. The sex-age-groups are determined by the rows existing in the ages and sexes tables.

The query takes two parameters specified on the first line of its SQL: [Date] of type DateTime and [DaysInYear] of type single precision floating point. [Date] is the date on which the tabulation is to be performed, and [DaysInYear] is the number of days the user wishes to define as one "year". This parameter can have a fractional part such as 0.25, but it this is not recommended because in

almost all cases the input data does not have dates specified to the precision of a fraction of a day (i.e. hours and minutes), and this additional precision in the duration of one year is therefore spurious and unwarranted. 365 days is the recommended duration of one year.

The SELECT statement begins by specifying the fields that are to be returned by the query: Sex, AgeGroup and a count of the ID field of the people table. Sex comes from the people table, AgeGroup from the ages table and the count() function aggregates and counts over the categories defined by the GROUP BY clause toward the end of the query: Sex, AgeGroup and StartAge. StartAge is redundant with AgeGroup and is included in the GROUP BY clause only to allow us to include it in the ORDER BY clause so that the resulting records can be sorted by sex and the starting age of each age group (a numeric value as opposed to the AgeGroup label which is text and does not sort in numeric order).

The FROM clause indicates that a Cartesian product is formed between the people and ages tables. This has the effect of pairing each person in the people table with *all* of the age groups in the ages table.

The WHERE clause restricts the records included in the result such that there is a *unique* record for each person; the one that satisfies the conditions: 1) that the person's current age (calculated on [Date]) falls within the time period defined by the days that begin and end (inclusive) the age group with which the record is associated, and 2) that the person has been born on or before [Date] and survives to at least [Date].

4.2 Explanation of query:

qrySelectFrequencyBySexAge_General_AllAges

AIM: To present the count of people alive on [Date] by sex and age for all categories of sex and age even if the count is zero.

It is often necessary to construct tables with the sex-age count of the living population for all categories of sex and age, and it is far easier to cut and paste a table containing a value for all categories of sex and age rather than having to fill in "by hand" the values transposing from the query results values only when the count is larger than zero. This query simply reports counts for all age-sex categories and makes the transposition of the counts to a presentation table easier; which is not a trivial concern when it is necessary to construct tables with tens or hundreds of age-sex-specific counts!

The SELECT statement begins by specifying the fields to be returned by the query: Sex, AgeGroup, and Frequency. The last is massaged slightly to insert a zero if a null value is encountered.

The FROM clause uses the JOIN syntax and indicates that a Cartesian product is formed between result of the sex-age-groups query (Example 3-2, above) and the qrySelectFrequencyBySexAge_General query matching on both sex and age group.

Finally, the ORDER BY clause sorts the resulting sex-age groups first by sex and then by age, again using the numeric value of StartAge, instead of AgeGroup, to

properly order by age group.

4.3 A Further Note on Counts and SQL

There are additional features of SQL that allow it to perform very powerful crosstab-like tabulations. These are variously implemented as CUBE and ROLLUP operators, but their essential function is the same; they all calculate n-wise tabulations on potentially very large datasets with the addition of a potentially large number of conditions that must be satisfied in order to be included in the tabulation. In light of these operators, SQL arguably provides the most powerful and flexible cross-tabulation capabilities available². Many Data Mining operations rely heavily on complex forms of the CUBE and ROLLUP operators. There is a vast literature on these topics, and the interested reader is encouraged to pursue it on their own.

•

² Dedicated crosstab engines can be much faster at the expense of the literally infinite flexibility offered by SQL.

5 Person-Years

Person years form the denominator of all event-exposure rates, and consequently calculating person years of exposure is of fundamental importance to Demographers. The calculation of person years is traditionally skipped over and assumed to be relatively trivial, but when one is confronted with such a calculation, numerous difficulties arise; including, over what time period should the calculation be made, how many days, hours and minutes are actually in a year, what do we do about months with different numbers of days and leap years, how are age groups defined, and finally, how do we construct reproducible, defensible, efficient, and reusable calculations that address all these concerns and produce a reasonable estimate of the person years exposed by various categories of the unit of analysis? Finally, given that the data are often stored in a relational database, how can we use SQL to perform the person years calculation in a generic, reusable way with user-defined parameters defining the time period covered by the calculation and the analyst's preference for the exact duration of one year? Example 5-1, below is an efficient solution to these challenges using standard SQL and one user-defined function.

qrySelectPersonYearsBySexAge_General:

```
PARAMETERS [Begin] DateTime, [End] DateTime, [DaysInYear] IEEESingle;
SELECT tblPeople.Sex, tblAges.AgeGroup,
Round((Sum(ftnIntervalIntersection([Begin],[End],[DOB]+[StartAge]*[DaysIn
Year],IIf(IsNull([DOD]),[DOB]+([EndAge]+1)*[DaysInYear]-
1,IIf([DOD]<[DOB]+([EndAge]+1)*[DaysInYear]-
1,[DOD],[DOB]+([EndAge]+1)*[DaysInYear]-1)])))/[DaysInYear],2) AS
PersonYears
FROM tblAges, tblPeople
WHERE (((tblPeople.DOB)<=[End]) AND ((tblPeople.DOD)>=[Begin])) OR
(((tblPeople.DOB)<=[End]) AND ((tblPeople.DOD) Is Null))
GROUP BY tblPeople.Sex, tblAges.AgeGroup, tblAges.StartAge
ORDER BY tblPeople.Sex, tblAges.StartAge;
```

ftnIntervalIntersection:

```
Public Function ftnIntervalIntersection( _
ByVal dteB1 As Long, _
ByVal dteE1 As Long, _
ByVal dteB2 As Long, _
ByVal dteE2 As Long) As Long
```

'Logic

' - Takes two intervals defined by their beginning and ending dates as input
' - For each interval, the begin date must precede the end date and the begin
' and end date cannot be the same
' - Valid time during the intervals is assumed to include all of the days
' spanned by the interval including the begin date but excluding the end
' date
' - The two intervals are compared and the number of days contained in any
' overlap is calculated

'Return values

' valid : >=0 the number of days that fall within both of the input intervals
' logic error : -1
' invalid input : 0

Example 5-1

'Zero results from either invalid input or no overlap; this prevents one invalid interval from killing a query that calls this function thousands of times, without invalidating the overall result

'Verify that the intervals are valid intervals

If (dteE1 - dteB1 >= 0) And (dteE2 - dteB2 >= 0) Then

'Intersect the intervals and return the length of the intersecting segment

'Left and Right defined by interval 1 and 2

'Add one to the result to account for the fact that the subtraction does not include the terminal day, and we want closed-closed interval arithmetic

'Left overlap

If (dteB1 < dteB2) And (dteE1 >= dteB2) And (dteE1 <= dteE2) Then

ftnIntervallIntersection = dteE1 - dteB2 + 1

'Left no overlap

Elseif (dteB1 < dteB2) And (dteE1 < dteB2) Then

ftnIntervallIntersection = 0

'Right overlap

Elseif (dteB1 >= dteB2) And (dteB1 <= dteE2) And (dteE1 > dteE2) Then

ftnIntervallIntersection = dteE2 - dteB1 + 1

'Right no overlap

Elseif (dteB1 > dteE2) And (dteE1 > dteE2) Then

ftnIntervallIntersection = 0

'2 contains 1

Elseif (dteB1 >= dteB2) And (dteE1 <= dteE2) Then

ftnIntervallIntersection = dteE1 - dteB1 + 1

'1 contains 2

Elseif (dteB1 <= dteB2) And (dteE1 >= dteE2) Then

ftnIntervallIntersection = dteE2 - dteB2 + 1

'Error in logic: the defined cases do not capture the temporal relationship between the two intervals

Else

ftnIntervallIntersection = -1

End If

'One of the intervals is not valid: i.e. it is of zero length or the end date precedes the begin date

Else

ftnIntervallIntersection = 0

End If

End Function

The primary reason the calculation of person years is complex is that it involves *temporal logic*. One must account for a number of *states* that an individual can occupy through time, perform logic to determine if the state occupied by an individual at a given point in time can be included in the accumulation of person years, and finally accumulate and present the total person years exposed for various categories of the unit of analysis – usually sex and age, at a minimum.

Because a temporal state is defined as a duration of time during which an attribute of interest does not change, it is most efficient to store only a representation of the events that cause an individual's state to change. In the framework of temporal states, an event is simply an instantaneous (on the time scale of interest) occurrence that results in a change of state for the individual(s) who experienced it. For the calculation of person years, the time scale is one day, the unit of analysis is people, and the events of interest are individuals' birthdays and the beginning and end of the period over which the

person year calculation is made. The birthdays designate the instant when individuals change state from one age to the next, and the beginning and end of the period over which the calculation is made encapsulate the duration over which we want an “observation” of person years exposed. In both cases the anchored duration over which a given state persists is called an *episode*. An episode has a defined (by events) beginning and ending and corresponds to the anchored duration of time during which a well-defined state associated with a meaningful *thing* persists.

The birthdays are stored in an abstract way in the ages table (tblAges) as unanchored durations of time that when compared to (or anchored by) individuals’ dates of birth identify the dates when the individuals experience a birthday and transition to the next age. In a generic and easily modified way, the ages table defines the *age-episodes* of individuals. The dates defining the period over which the calculation is to be made are provided by the analyst and define and anchor the *observation-episode* of interest.

By comparing the age-episodes to the observation-episode one can determine how much of each age-episode of each individual falls within *or overlaps* the observation-episode. It is then straightforward to aggregate by sex and age-group and take the sum of the overlaps to yield the total person years exposed by sex and age.

The overlap logic is accomplished by the user-defined function `ftnIntervallIntersection`; taking as arguments the beginning and end dates for the two episodes being compared, and returning the number of *days* that the two episodes overlap, or equivalently, the number of *dates* shared by the two episodes.

Table 8 through Table 10, below, present the results of running the `qrySelectPersonYearsBySexAge_General` query on the Big People and Small People tables. In Table 8, total person years exposed by sex and age are calculated over the period 1/1/2000 to 1/1/2100 for the Big People table. All sex-age groups are presented, even when there are no person years exposed, so that the results can be easily copied (cut and pasted) to a presentation table. The numbers look plausible, and when combined with an understanding of the construction of the Big People table, a little reflection convinces us that the totals of sex-age-specific person years presented are accurate.

**Table 8: qrySelectPersonYearsBySexAge_General:
Run on the Big People Table over the Period 1/1/2000 to 1/1/2100**

Sex	AgeGroup	PersonYears
Female	0	620.43
Female	1-4	2383.61
Female	5-9	2787.85
Female	10-14	2561.85
Female	15-19	2325.37
Female	20-24	2089.59
Female	25-29	1848.84
Female	30-34	1597.3
Female	35-39	1344.67
Female	40-44	1092.37
Female	45-49	855.07

Sex	AgeGroup	PersonYears
Female	50-54	721.99
Female	55-59	656.94
Female	60-64	586.11
Female	65-69	511.13
Female	70-74	430.3
Female	75-79	369.25
Female	80-84	284.26
Female	85-89	209.08
Female	90-94	114.4
Female	95-99	34.79
Female	100-104	0
Female	105-109	0
Female	110-114	0
Female	115+	0
Male	0	570.63
Male	1-4	2200.75
Male	5-9	2537.87
Male	10-14	2314.19
Male	15-19	2100.96
Male	20-24	1887.06
Male	25-29	1678.11
Male	30-34	1479.97
Male	35-39	1282.9
Male	40-44	1085.51
Male	45-49	873.12
Male	50-54	706.7
Male	55-59	621.85
Male	60-64	542.8
Male	65-69	467.87
Male	70-74	398.81
Male	75-79	309.96
Male	80-84	245.06
Male	85-89	170.33
Male	90-94	115.11
Male	95-99	44.82
Male	100-104	0.07
Male	105-109	0
Male	110-114	0
Male	115+	0

Table 9 and Table 10 present the same query run on the Small People table over the periods 1/1/2000 to 1/1/2100 and 1/1/1980 to 12/31/2000 respectively. Because there are only four records in the Small People table, it is easy to verify that the results of the query are accurate. Table 11 is the result of running the same query over the period 1/1/1980 to 12/31/2000 using the Small Ages table to produce the calculation using fewer, wider (more aggregated) age groups.

Table 9: qrySelectPersonYearsBySexAge_General:

Run on the Small People Table over the Period 1/1/2000 to 1/1/2100

Sex	AgeGroup	PersonYears
Female	0	1.54
Female	1-4	4
Female	5-9	5
Female	10-14	5
Female	15-19	5
Female	20-24	5

Sex	AgeGroup	PersonYears
Female	25-29	5
Female	30-34	5
Female	35-39	5
Female	40-44	5
Female	45-49	5
Female	50-54	5
Female	55-59	5
Female	60-64	5
Female	65-69	5
Female	70-74	5
Female	75-79	5
Female	80-84	0.59
Female	85-89	0
Female	90-94	0
Female	95-99	0
Female	100-104	0
Female	105-109	0
Female	110-114	0
Female	115+	0
Male	0	1
Male	1-4	4
Male	5-9	5
Male	10-14	5
Male	15-19	5
Male	20-24	9.99
Male	25-29	10
Male	30-34	10
Male	35-39	10
Male	40-44	10
Male	45-49	10
Male	50-54	6.04
Male	55-59	5
Male	60-64	5
Male	65-69	5
Male	70-74	1.05
Male	75-79	0
Male	80-84	0
Male	85-89	0
Male	90-94	0
Male	95-99	0
Male	100-104	0
Male	105-109	0
Male	110-114	0
Male	115+	0

**Table 10: qrySelectPersonYearsBySexAge_General:
Run on the Small People Table over the Period 1/1/1980 to 12/31/2000**

Sex	AgeGroup	PersonYears
Female	0	1.54
Female	1-4	0
Female	5-9	0
Female	10-14	0
Female	15-19	0
Female	20-24	0

Sex	AgeGroup	PersonYears
Female	25-29	0
Female	30-34	0
Female	35-39	0
Female	40-44	0
Female	45-49	0
Female	50-54	0
Female	55-59	0
Female	60-64	0
Female	65-69	0
Female	70-74	0
Female	75-79	0
Female	80-84	0
Female	85-89	0
Female	90-94	0
Female	95-99	0
Female	100-104	0
Female	105-109	0
Female	110-114	0
Female	115+	0
Male	0	2
Male	1-4	4
Male	5-9	5
Male	10-14	5
Male	15-19	5
Male	20-24	1.02
Male	25-29	0
Male	30-34	0
Male	35-39	0
Male	40-44	0
Male	45-49	0
Male	50-54	0
Male	55-59	0
Male	60-64	0
Male	65-69	0
Male	70-74	0
Male	75-79	0
Male	80-84	0
Male	85-89	0
Male	90-94	0
Male	95-99	0
Male	100-104	0
Male	105-109	0
Male	110-114	0
Male	115+	0

**Table 11 : qrySelectPersonYearsBySexAge_General:
Run on the Small People Table over the Period 1/1/1980 to 12/31/2000
Using the Small Ages Age-Groups**

Sex	AgeGroup	PersonYears
Female	0	1.54
Female	1-4	0
Female	5-19	0
Female	20-49	0
Female	50+	0

Sex	AgeGroup	PersonYears
Male	0	2
Male	1-4	4
Male	5-19	15
Male	20-49	1.02
Male	50+	0

The next two sections contain a detailed examination of the person years query and its user-defined overlap function and should be skipped by the casual reader.

5.1 Explanation of function: `ftnIntervalIntersection`

The commented code for the function `ftnIntervalIntersection` is included in Example 5-1. This function calculates the duration of the overlap (in days) between two time intervals (anchored durations). It takes as arguments the beginning and ending dates of the two intervals and returns the number of days they overlap as an integer.

It first checks that the intervals it is passed are valid and of at least zero duration; or that the begin date precedes or is identical to the end date.

If the intervals are valid, the two intervals must be related to each other in one of six ways:

1. **left overlap**: the first interval begins before and ends during the second interval,
2. **left no overlap**: the first interval begins and ends before the second interval,
3. **right overlap**: the second interval begins during and ends after the first interval,
4. **right no overlap**: the second interval begins and ends after the first,
5. **2 contains 1**: the second interval contains the first interval, and
6. **1 contains 2**: the first interval contains the second.

These six possibilities are a subset of what are known as *Allen's Operators* (Allen 1983) but are sufficient for our purposes. The overlapping portion of the intervals is calculated in a different way depending on which of the six relationships the intervals share, and the duration of that overlap in days is returned by the function; please see Example 5-1 for the details of how the overlap is calculated in each case. If none of the six relationships describes the intervals an error value of '-1' is returned. The function returns '0' for intervals that do not overlap or when one or intervals are invalid (begin after they end).

5.2 Explanation of query:

`qrySelectPersonYearsBySexAge_General`

AIM: To present the person years exposed between [Beginning] and [End] (dates) by sex and age for all categories of sex and age even if the total is zero.

The query takes as date type parameters [Begin] and [End] specifying the dates

on which the time interval for the calculation starts and ends and [DaysInYear] as a single precision floating point number specifying the number of days in one year.

The SELECT clause specifies three fields from the Cartesian product of tblAges and tblPeople: Sex, AgeGroup, and PersonYears, discussed below. The Cartesian product pairs every age group from the ages table with every individual in the people table. In this way every individual has the potential to contribute person-time to the total person years exposed in every age group. The WHERE clause insures that the individual's life overlaps with the analysis interval at all, and the interval intersection function (above) determines how much of each age interval for each person overlaps with the analysis interval.

The WHERE clause specifies some temporal logic to insure that the only records eligible for inclusion are those for which some part of the individual's life overlaps with the analysis interval; formally, that the following condition holds:

```
((tblPeople.DOB)<=[End]) AND ((tblPeople.DOD)>=[Begin]) OR  
((tblPeople.DOB)<=[End]) AND (tblPeople.DOD) Is Null).
```

In other words, those for which:

(the individual is born before the end of the analysis interval) AND (the individual dies after the beginning of the analysis interval)

OR

(the individual is born before the end of the analysis interval) AND (the individual has not yet died by the end of the analysis interval)

The resulting records that remain after these WHERE conditions are met form the basis for the person years calculation. Each of these records consists of an age group identifier (string describing the ages included in the age group), the beginning [StartAge] and ending [EndAge] ages of the age group in years since birth, and an individual's sex, date of birth [DOB], and date of death [DOD] (null if the individual is still alive at the last time observed).

The [DOB], [DOD], [StartAge], and [EndAge] fields are used to determine the date of all of the birthdays that an individual has that result in a change from one age group to the next, as defined by the ages table. Each record corresponds to the unique matching of an age group with an individual, and each individual is matched with **all** age groups.

Each of these age group-individual records is then fed to the interval intersection function passing it the beginning and end of the analysis interval, [Begin] and [End] as the definition of the 'first' interval, and passing it the dates on which the individual started and stopped living through the age group as the 'second' interval. The interval intersection function returns the number of days that the individual lived in the age group during the analysis interval and stores it in the [PersonYears] field.

The records are then grouped by sex, age group, and the start age of each age group. Over each of these groups the number of days stored in the [PersonYears] field is summed and divided by the value of [DaysInYear] (passed to the query as a parameter) to yield the total number of years exposed in each

sex-age group.

Following is a commented copy of the code fragment that is called to calculate the overlap between the age group and the analysis interval; what populates the [PersonYears] field in the Person Years query.

Sum over sex-age groups.

Sum(

Call the interval intersection function.

 fnIntervallIntersection(

The 'first' episode is the analysis interval defined by the [Begin] and [End] date values passed to the query as parameters.

 [Begin],[End],

The 'second' episode is the age interval as defined by the individual's date of birth, date of death, and the ages specified in the ages table. The beginning date of the age interval is the individual's date of birth plus the start age of the age group multiplied times the number of days in a year, [DaysInYear] passed to the query as a parameter.

 [DOB]+[StartAge]*[DaysInYear],

The end of the age interval is complicated. If the individual has not died, the end of the age interval is the individual's date of birth plus the end age of the age group multiplied times the number of days in one year. The '+1' and '-1' account for the fact that the end age for the age group is specified as the last completed age, not the age of the next older age group, and we do not count the last day an age interval because we count the first. If the individual has died, if the date of death is before the end of the age interval, the end of the age interval is the individual's date of death, otherwise, the end of the age interval is the individual's date of birth plus the end age of the age group multiplied times the number of days in one year, same issue with the '+1' and '-1'.

 IIf(IsNull([DOD]), [DOB]+([EndAge]+1)*[DaysInYear]-1,

 IIf([DOD]<[DOB]+([EndAge]+1)*[DaysInYear]-1, [DOD], [DOB]+([EndAge]+1)*[DaysInYear]-1))

)

The sum of the overlap days in each sex-age group is divided by the number of days in one year to yield the person years exposed in each sex-age group.

)/[DaysInYear])

•

6 Event-Exposure Rates

Dividing the number of events occurring to a defined group of individuals by the number of person years that they were exposed to the risk of the event yields the staple measure of Demography: the event-exposure rate. Amongst many others, standard mortality and fertility rates are calculated in this way.

Now that we have SQL queries to calculate event counts and person years exposed, it is an easy matter to combine them into a single query that calculates an event-exposure rate. Such a query is presented in Example 6-1.

qrySelectMortalityRate_General:

```
SELECT qrySelectSexAgeGroups.Sex, qrySelectSexAgeGroups.AgeGroup,
       qrySelectDeathsBySexAge_General.Deaths,
       qrySelectPersonYearsBySexAge_General.PersonYears,
       Round((IIf([PersonYears]=0 Or
                  IsNull([Deaths]),0,1000*([Deaths]/[PersonYears])),2) AS MortalityRate
FROM (qrySelectSexAgeGroups
      LEFT JOIN qrySelectDeathsBySexAge_General ON
        (qrySelectSexAgeGroups.AgeGroup =
         qrySelectDeathsBySexAge_General.AgeGroup) AND
        (qrySelectSexAgeGroups.Sex = qrySelectDeathsBySexAge_General.Sex))
      LEFT JOIN qrySelectPersonYearsBySexAge_General ON
        (qrySelectSexAgeGroups.AgeGroup =
         qrySelectPersonYearsBySexAge_General.AgeGroup) AND
        (qrySelectSexAgeGroups.Sex = qrySelectPersonYearsBySexAge_General.Sex)
ORDER BY qrySelectSexAgeGroups.Sex, qrySelectSexAgeGroups.StartAge;
```

Example 6-1

The results of running this query on both the Big and Small People tables using both the Big and Small Ages tables over various periods are displayed in Table 12 through Table 14. The age group, number of deaths, number of person years and finally the mortality rate per 1,000 is displayed. This query provides a re-usable, clean, efficient and flexible way to extract mortality rates directly from a relational database without the need for any external logic or processing. Calculation of any event-exposure rate can be performed in an analogous way.

Table 12: qrySelectMortalityRate_General:

Run on the Small People Table over the Period 1/1/1980 to 7/15/2080

Using the Small Ages Age-Groups

Sex	AgeGroup	Deaths	PersonYears	MortalityRate
Female	0	1	1.54	649.35
Female	1-4		4	0
Female	5-19		15	0
Female	20-49		30	0
Female	50+	1	30.59	32.69
Male	0		2	0
Male	1-4		8	0
Male	5-19		30	0
Male	20-49		60	0
Male	50+	2	22.08	90.58

Table 13: qrySelectMortalityRate_General:

Run on the Big People Table over the Period 1/1/2000 to 12/31/2099

Using the Small Ages Age-Groups

Sex	AgeGroup	Deaths	PersonYears	MortalityRate
Female	0	3	620.42	4.84
Female	1-4	25	2383.57	10.49
Female	5-19	84	7674.93	10.94
Female	20-49	198	8827.57	22.43
Female	50+		3917.84	0
Male	0	9	570.62	15.77
Male	1-4	23	2200.72	10.45
Male	5-19	96	6952.91	13.81
Male	20-49	162	8286.44	19.55
Male	50+		3622.98	0

Table 14: qrySelectMortalityRate_General:

Run on the Big People Table over the Period 1/1/2000 to 12/31/2099

Using the Big Ages Age-Groups

Sex	AgeGroup	Deaths	PersonYears	MortalityRate
Female	0	3	620.42	4.84
Female	1-4	25	2383.57	10.49
Female	5-9	24	2787.8	8.61
Female	10-14	32	2561.79	12.49
Female	15-19	28	2325.33	12.04
Female	20-24	35	2089.55	16.75
Female	25-29	29	1848.79	15.69
Female	30-34	34	1597.25	21.29
Female	35-39	36	1344.62	26.77
Female	40-44	34	1092.33	31.13
Female	45-49	30	855.03	35.09
Female	50-54		721.96	0
Female	55-59		656.9	0
Female	60-64		586.07	0
Female	65-69		511.09	0
Female	70-74		430.25	0
Female	75-79		369.21	0
Female	80-84		284.21	0
Female	85-89		209.04	0
Female	90-94		114.36	0
Female	95-99		34.75	0
Female	100-104		0	0
Female	105-109		0	0
Female	110-114		0	0
Female	115+		0	0
Male	0	9	570.62	15.77
Male	1-4	23	2200.72	10.45
Male	5-9	36	2537.83	14.19
Male	10-14	28	2314.16	12.1
Male	15-19	32	2100.92	15.23
Male	20-24	25	1887.02	13.25
Male	25-29	31	1678.07	18.47
Male	30-34	26	1479.94	17.57
Male	35-39	24	1282.86	18.71
Male	40-44	26	1085.47	23.95
Male	45-49	30	873.07	34.36

Male	50-54		706.65	0
Male	55-59		621.81	0
Male	60-64		542.75	0
Male	65-69		467.83	0
Male	70-74		398.77	0
Male	75-79		309.91	0
Male	80-84		245.02	0
Male	85-89		170.3	0
Male	90-94		115.07	0
Male	95-99		44.79	0
Male	100-104		0.07	0
Male	105-109		0	0
Male	110-114		0	0
Male	115+		0	0

6.1 Explanation of query:

qrySelectMortalityRate_General

AIM: To present the event-exposure mortality rate between [Beginning] and [End] (dates) by sex and age for all categories of sex and age even if the mortality rate is zero.

The query takes as date type parameters [Begin] and [End] specifying the dates on which the time interval for the calculation starts and ends and [DaysInYear] as a single precision floating point number specifying the number of days in one year. These are passed through to the Count Deaths and Total Person Years queries.

The FROM clause specifies that the Cartesian product from which the result set will be drawn consists of all records resulting from the Sex Age Groups query paired with all records resulting from the sex-age-specific Count Deaths query, and further, all of those paired with all records resulting from the sex-age-specific Total Person Years query. The overall Cartesian product is restricted such that only records with identical sex and age group values from all three input queries are retained. The result is one record for each sex-age group as defined by the sexes and ages tables. Pairing both the Count Deaths and Total Person Years queries with the Sex-Age-Groups query insures that a record exists for all sex-age combinations and not just those for which values exist in the Count Deaths and Total Person Years queries.

The SELECT clause specifies that the sex and age group fields be displayed from the Sex-Age-Groups query, that the count of deaths be displayed from the Count Deaths query, that the total person years be displayed from the Total Person Years query, and finally that the mortality rate per 1,000 is calculated from the count of deaths and total person years and rounded to two digits after the decimal point and displayed as the [MortalityRate] field.

•

7 Person-Time Format Tables

One of the most powerful and certainly the most flexible technique available for statistical analysis of longitudinal data is the Discrete Time Event History method (Allison 1982, 1984; Rodríguez 2002). The observed life span of each unit of analysis is divided into consecutive time intervals such as months or years and one record is created for each person-time interval. At the beginning of each interval, both the time invariant and time-varying quantities of interest are defined and recorded with respect to the unit of analysis to which the time interval pertains. These are values of the risk factors associated with some outcome(s) of interest that may occur during the time interval. Whether or not the outcome(s) of interest occur during each time interval is also recorded. The result is a dataset that can be analyzed using logistic regression. Both left and right censoring are handled easily by not including person-time intervals when the unit of analysis is not observed *throughout*, and both time-varying and time invariant cofactors are handled in the same consistent way by defining their values at the beginning of each person-time interval. The result of an event history analysis is the probability of the outcome(s) of interest as a function of both time-varying and time-invariant cofactors. Because the analysis is done within a statistical framework, standard statistical tests can be performed to rule out the null hypothesis that the cofactors of interest have no effect, to test for the interaction of cofactors, to examine time trends and duration effects, and to “control” for various factors.

In order to perform Event History-based analyses of temporal data, person-time interval datasets are necessary. These are relatively difficult to construct using standard approaches because they necessitate looping through both time and the units of analysis. This requires some form of programming, and even though the programming is trivial, mistakes often occur, especially with respect to the proper specification of the entry and exit from observation criteria. SQL greatly simplifies the creation of person-time interval datasets by collapsing the whole process into one straightforward query.

The query needs two tables as inputs, the Dates table containing one row for each time interval required for the analysis, and a table describing the units of analysis with one row for each of the units of analysis. The records in the Dates table must each correspond to an interval of time with the *same duration*, but they need not be consecutive intervals, although they usually are.

The query forms the Cartesian product of the two tables *pairing each observed time interval with each unit of analysis*. The result set is restricted to only include person-time interval records for which the person was *alive and observed*. The values of each of the cofactors is calculated and stored in new fields, and whether or not the outcomes of interest occur is also recorded in new fields. The fields required in the final dataset are selected and all of the new person-time interval records are written to a new table. A query of this type is presented in Example 7-1. This “Make Person Years” query creates a table with one record per person-year over the period of years contained in the dates table (tblDates).

qryMakePersonYears:

```
PARAMETERS [DaysInYear] IEEESingle;
SELECT tblPeople.ID, tblPeople.Sex, tblPeople.DOB, tblPeople.DOD,
Round(((1+[BeginDate]-[DOB])/[DaysInYear]),2) AS AgeAtBeginning, If(((Not
IsNull([DOD])) And ([DOD]>=[BeginDate]) And ([DOD]<=[EndDate])),True,False)
AS Died, tblDates.BeginDate, tblDates.EndDate INTO tblPersonYears
FROM tblDates, tblPeople
WHERE (((If(([DOB]<=[BeginDate]) And (IsNull([DOD]) Or
([DOD]>=[BeginDate])),True,False))=True))
ORDER BY tblPeople.ID, tblDates.BeginDate;
```

Example 7-1

The abbreviated result of running the Make Person Years query on the Small People table using a Dates table with records for the years 1970 through 2084 is shown Table 15. The complete set of person years for individuals 1 and 2 are presented along with the first two person years lived by individual 3. The rest of the result set is omitted for the sake of brevity. Each person year contains fields describing the individual’s ID, sex, date of birth, date of death, age at the beginning of the time interval, whether or not they died during the interval, the date of the beginning of the interval, and the date of the end of the interval. This is a very simple data structure that could be used (with an appropriately large sample) to produce sex-age-specific estimates of the annual *hazard* (duration-specific probability) of death. A conventional life table could be constructed from those values converted directly to standard life table age-specific probabilities of death (${}_nq_x$) values. Additionally with this technique, the null hypothesis that male and female probabilities of death are equal at each age could be tested, and the null hypothesis that age-specific mortality rates do not change from one historical period to the next could be tested – among many other possibilities.

Table 15: qryMakePersonYears:

Run on the Small People Table over the Period 1/1/1970 to 12/31/2084

ID	Sex	DOB	DOD	AgeAtBeginning	Died	BeginDate	EndDate
1	Male	1/1/2000	12/31/2050	0	0	1/1/2000	12/31/2000
1	Male	1/1/2000	12/31/2050	1.01	0	1/1/2001	12/31/2001
1	Male	1/1/2000	12/31/2050	2.01	0	1/1/2002	12/31/2002
1	Male	1/1/2000	12/31/2050	3.01	0	1/1/2003	12/31/2003
1	Male	1/1/2000	12/31/2050	4.01	0	1/1/2004	12/31/2004
1	Male	1/1/2000	12/31/2050	5.01	0	1/1/2005	12/31/2005
1	Male	1/1/2000	12/31/2050	6.01	0	1/1/2006	12/31/2006
1	Male	1/1/2000	12/31/2050	7.01	0	1/1/2007	12/31/2007
1	Male	1/1/2000	12/31/2050	8.01	0	1/1/2008	12/31/2008
1	Male	1/1/2000	12/31/2050	9.01	0	1/1/2009	12/31/2009
1	Male	1/1/2000	12/31/2050	10.01	0	1/1/2010	12/31/2010
1	Male	1/1/2000	12/31/2050	11.01	0	1/1/2011	12/31/2011
1	Male	1/1/2000	12/31/2050	12.01	0	1/1/2012	12/31/2012
1	Male	1/1/2000	12/31/2050	13.01	0	1/1/2013	12/31/2013
1	Male	1/1/2000	12/31/2050	14.01	0	1/1/2014	12/31/2014
1	Male	1/1/2000	12/31/2050	15.01	0	1/1/2015	12/31/2015
1	Male	1/1/2000	12/31/2050	16.01	0	1/1/2016	12/31/2016
1	Male	1/1/2000	12/31/2050	17.02	0	1/1/2017	12/31/2017
1	Male	1/1/2000	12/31/2050	18.02	0	1/1/2018	12/31/2018
1	Male	1/1/2000	12/31/2050	19.02	0	1/1/2019	12/31/2019
1	Male	1/1/2000	12/31/2050	20.02	0	1/1/2020	12/31/2020
1	Male	1/1/2000	12/31/2050	21.02	0	1/1/2021	12/31/2021

ID	Sex	DOB	DOD	AgeAtBeginning	Died	BeginDate	EndDate
1	Male	1/1/2000	12/31/2050	22.02	0	1/1/2022	12/31/2022
1	Male	1/1/2000	12/31/2050	23.02	0	1/1/2023	12/31/2023
1	Male	1/1/2000	12/31/2050	24.02	0	1/1/2024	12/31/2024
1	Male	1/1/2000	12/31/2050	25.02	0	1/1/2025	12/31/2025
1	Male	1/1/2000	12/31/2050	26.02	0	1/1/2026	12/31/2026
1	Male	1/1/2000	12/31/2050	27.02	0	1/1/2027	12/31/2027
1	Male	1/1/2000	12/31/2050	28.02	0	1/1/2028	12/31/2028
1	Male	1/1/2000	12/31/2050	29.02	0	1/1/2029	12/31/2029
1	Male	1/1/2000	12/31/2050	30.02	0	1/1/2030	12/31/2030
1	Male	1/1/2000	12/31/2050	31.02	0	1/1/2031	12/31/2031
1	Male	1/1/2000	12/31/2050	32.02	0	1/1/2032	12/31/2032
1	Male	1/1/2000	12/31/2050	33.03	0	1/1/2033	12/31/2033
1	Male	1/1/2000	12/31/2050	34.03	0	1/1/2034	12/31/2034
1	Male	1/1/2000	12/31/2050	35.03	0	1/1/2035	12/31/2035
1	Male	1/1/2000	12/31/2050	36.03	0	1/1/2036	12/31/2036
1	Male	1/1/2000	12/31/2050	37.03	0	1/1/2037	12/31/2037
1	Male	1/1/2000	12/31/2050	38.03	0	1/1/2038	12/31/2038
1	Male	1/1/2000	12/31/2050	39.03	0	1/1/2039	12/31/2039
1	Male	1/1/2000	12/31/2050	40.03	0	1/1/2040	12/31/2040
1	Male	1/1/2000	12/31/2050	41.03	0	1/1/2041	12/31/2041
1	Male	1/1/2000	12/31/2050	42.03	0	1/1/2042	12/31/2042
1	Male	1/1/2000	12/31/2050	43.03	0	1/1/2043	12/31/2043
1	Male	1/1/2000	12/31/2050	44.03	0	1/1/2044	12/31/2044
1	Male	1/1/2000	12/31/2050	45.04	0	1/1/2045	12/31/2045
1	Male	1/1/2000	12/31/2050	46.04	0	1/1/2046	12/31/2046
1	Male	1/1/2000	12/31/2050	47.04	0	1/1/2047	12/31/2047
1	Male	1/1/2000	12/31/2050	48.04	0	1/1/2048	12/31/2048
1	Male	1/1/2000	12/31/2050	49.04	0	1/1/2049	12/31/2049
1	Male	1/1/2000	12/31/2050	50.04	-1	1/1/2050	12/31/2050
2	Female	1/1/2000	7/15/2000	0	-1	1/1/2000	12/31/2000
3	Male	1/1/1980	12/31/2050	0	0	1/1/1980	12/31/1980
3	Male	1/1/1980	12/31/2050	1.01	0	1/1/1981	12/31/1981
?	?	?	?	?	?	?	?

7.1 Explanation of query: qryMakePersonYears

AIM: To create a table consisting of one record for each year (or more generally any arbitrary time interval) that each person lives between [Beginning] and [end] (dates), and to record in each record the value of relevant variables describing the person *at the beginning* of the year.

The query takes as a parameter the number of days in one year as a single precision floating point number. Additionally, the analyst must supply the dates table consisting of one record for each time interval to be included in the calculation. The records must all correspond to time intervals of equal duration, because the duration of exposure to the outcome events is the temporal referent for the hazard rate that is the ultimate product of the Event History Analysis. The sample database from which the examples are drawn contains a simple user-defined function to populate the Dates table with an arbitrary number of time intervals of either day, month, or year duration.

The SELECT clause forms the Cartesian product between the Dates and People table pairing each time interval with each person. Fields included in the result set include the person's ID, sex, date of birth, date of death, date of the

beginning of the interval, and date of the end of the interval. Calculated fields include the person's age at the beginning of the interval (making use of the number of days in one year) and a boolean variable indicating whether or not the person died during the interval.

The WHERE clause restricts the records in the result set to include only those for which the person was alive at the beginning of the time interval.

The result is a dataset containing one record for each person-time interval lived in the People table corresponding to a time interval included in the dates table. A collection of both time-varying and time invariant attributes of the people are defined at the beginning of each person-time interval, and whether or not the person died during the interval is recorded.

•

8 Summary

This communication provides a very basic introduction to the Relational Model of Data and the Standard Query Language used to manage data in a Relational Database. Illustrative Examples are provided using the most "standard" version of SQL commercially available, that found in IBM's DB2 Database Management System.

After covering the basics, various queries useful to Demographers are discussed and described in detail. Because of its ease of use and widespread availability, Microsoft's Access database is used to construct and run all of the Demographically interesting examples.

Both the DB2 script that runs the illustrative examples and the Access database with the Demography-related examples are available for download as part of this communication. If you are part of an educational or research organization, you can acquire a free copy of IBM's DB2 Database Management System through IBM's Scholar's program, see <http://www.ibm.com/db2>. Microsoft Access is a standard part of Microsoft's Office productivity suite, or it can be purchased as a standalone product.

•

9 Bibliography

- Allen, J. F. 1983. "Maintaining Knowledge about Temporal Intervals." *Communications of the ACM*, 26(11 November 1983):832-843.
- Allison, P. 1982. "Discrete-time Methods for the Analysis of Event Histories." Pp. 61-98 in *Sociological Methodology*, edited by S. Leinhardt. San Francisco: Jossey-Bass.
- . 1984. Event History Analysis. Regression for Longitudinal Event Data. Beverly Hills: Sage.
- Codd, E. F. 1970. "A Relational Model of Data for Large Shared Data Banks." *Communications of the ACM*, 13(6):377-387.
- . 1990. The Relational Model for Database Management Version 2. Reading Massachusetts: Addison-Wesley.
- Date, C. J. 2000. An Introduction to Database Systems, 7th ed. Reading Massachusetts: Addison-Wesley.
- Rodríguez, G. 2002. "Lecture Notes for Generalized Linear Statistical Models". <http://data.princeton.edu/wws509>. October, 2002.