第22章 时态数据库

22.1 引言

注意：本章的原作者为Hugh Darwen。

简单地说，时态数据库是包含历史数据或同时也包含当前数据的数据库。自从70年代中期以来人们就一直在研究时态数据库。一些极端的观点认为在这种数据库中只会进行数据插入，而从不删除或更新（参见上一章关于数据仓库的讨论），数据库中只有历史数据。另外一种极端的观点认为它是一种快照式数据库㊀，只包含当前数据，而且只是当数据不正确时才会进行删除或更新（换句话说，快照式数据库就是人们通常所指的数据库，而根本不是时态数据库）。

以图3-8中的供应商-零件数据库为例，它就是一个快照式数据库，其中显示供应商S1的当前状态为20。而在时态数据库中，可能不仅包含当前状态为20，而且包含自7月1日起状态都为20，从4月5日到6月30日状态为15之类的信息。

在快照式数据库中，快照的时间一般是“现在”（即观察数据库的时刻）。就算快照的时间不是“现在”，也不会影响数据的管理和使用方式。不过，在时态数据库中，数据的管理和使用方式与快照式数据库有很大的不同，这也是本章要讨论的内容。

时态数据库的特点当然就是时间，因此，对时态数据库的研究大多是对时间本质的研究。下面给出一些研究的问题：

- 时间是否有开始和结束（哲学问题）；
- 时间是连续的还是离散的（科学问题）；
- 如何最好地描述“现在”这个概念（心理学问题）；

不过这些问题与数据库没有什么关系，我们不作深入探讨，而只是作一些合理的假设，这样可以精力集中在需要解决的问题上。需要注意的是，对时态的某些研究已经不再局限于数据库的范畴，而扩展到其它领域（不过我们还是在本章中继续按原有约定来谈论“时态”码、“时态”操作符、“时态”关系，等等）。

在以上的这些研究中，我们只会去关注那些我们认为有意义的重要内容（换句话说，就是取其精华）。不过，要注意我们将要探讨的技术几乎没有在任何商业软件中使用，可能是出于以下原因：

正是在不久之前磁盘存储已经变得十分便宜，使得海量历史数据的存储成为可能。不过，正如第21章中所言，“数据仓库”日趋流行，因此用户会日益面临时态数据库问题，并希望寻求解决方案。

尽管我们探讨的技术尚未完全在商业软件中使用，它们与现有产品的结合——其是SQL产品——显得前景暗淡。另外，大多数销售商正尽力提供对象/关系支持（参见第25章）。

㊀ 与第9章所提到的快照概念无关。

研究机构采用不同的最佳方法来解决这些问题（这种不一致的方法也会影响到销售商）。有的研究人员倾向于一种十分特定的方法——和关系理论有些脱离——来处理时态数据，并遗留一些未解决的问题（参见 [22.4]）。另外一些研究人员则倾向于提供更通用的操作，如果需要的话，以在此基础上开发特定方法，这种方法符合关系理论（参见 [22.3]）。无庸置疑，我们采用后一种方法。

22.2 时态数据

如果说数据是事实信息的一种编码表达形式，那么时态数据就是与时间相关事实信息的编码表达形式。按上面的定义，在时态数据库中所有的数据都是时态数据，每条记录中都包含时间信息。时态关系中的每个元组中至少包含一个时间戳（即标题中至少包含一个时间戳类型的属性）。时态关系变量（temporal relvar）的标题是时态关系，（关系型）时态数据库中的所有关系变量都是时态关系变量。注意：我们故意不在这里定义什么是“时间戳类型”的数据，而留待第22.3~22.5节中讨论。

在给出了“时态数据库”的精确定义（按最严格的形式）之后，我们发现这个概念并没有什么实际用处！因为即使数据库中的关系变量都是时态的，从数据库中还是可以导出非时态的关系（比如查询结果）。例如，我们可以从时态数据库中“找到曾经雇佣的全体雇员名单”，但是查询结果本身并非时态关系。这就是一个很奇怪的 DBMS，很显然它不是关系型的，但却可以从中查出不能保存在数据库中的一些结果。

因此在本章中，我们把时态数据库定义为“包含某些时态数据，同时也可以包含非时态数据”的数据库。本章的各节是这样安排的：

- 在本节的剩余部分以及第 22.3节介绍背景知识；第 22.3节中还特别阐述为什么时态数据需要特殊处理；
- 第22.4节和第22.5节引入间隔（interval）作为表述时间戳数据的简便方法。第 22.6节和第22.7节讨论间隔上的各种标量和聚集操作符。
- 第22.8节引入时态关系上的一些重要关系操作符。
- 第22.9节讨论时态数据完整性检验问题。第 22.10节讨论相应的更新问题。
- 最后在第22.11节给出一些相关的数据库设计思想，第 22.12节是全章小结。

注意：读者必须明白——除了第22.5节中间隔生成操作符之外——本章中提到的所有操作符都只是一些简记符号，也就是说，可以用一种完备的关系语言如 Tutorial D来表述它们。

一些基本概念和问题

首先我们要纠正在自然语言中人们对“带有时间戳的语句”的阐述方法，下面有三个例子：

1) 在1999年7月1日委任供应商S1（即签订合同）。
2) 自从1999年7月1日与供应商S1有合同关系。
3) 从1999年7月1日到现在为止与供应商S1有合同关系。

每个句子中都包含了两项：供应商“S1”和时间戳“1999年7月1日”，它们都可以作为当前数据库的快照。句子中的黑体字，在、自从、从……到，意义都不同。

尽管上面三个句子的表述形式不同，它们却基本上表示相同的语义。实际上，我们一般认为第二句和第三句是等价的，而第一句则未必和其他两句等价。因为：

- 第一句清楚地表明了在1999年6月30日与供应商S1没有合同关系，而在第二句中则无法知道这一点。
- 假设现在是2000年9月25日，那么第二句就清楚地表明了从1999年7月1日到2000年9月25日都和供应商S1有合同关系，而在第一句中则无法知道这一点。

因此第一句和第二句不等价。

在快照数据库中常常会出现诸如“委任日期”之类的元组，以及类似第二句或第三句的阐述。如果事实情况如第二句所述，就应该把第一句改为“在1999年7月1日委任供应商S1，并与其保持合同关系至今”；而如果事实情况如第一句所述，则应该把第二句改为“在1999年6月30日与供应商S1没有合同关系，自从1999年7月1日与供应商S1有合同关系”。

第一句表述了事件发生的时刻，而第二句和第三句则表述了状态持续的时间间隔。我们特意选取了上面这个例子，从事件发生的时刻可以推断出后续的状态：因为“在1999年7月1日委任供应商S1，并与其保持合同关系至今”，所以“自从1999年7月1日与供应商S1有合同关系”。在第22.3节中我们会看到，传统数据库技术可以很好地处理时刻（事件发生的时刻），却不能很好地处理时间间隔（状态持续的时间间隔）。

第二句尽管在逻辑上与第三句等价，但表达形式显然不同。特定情况下，第二句不能用在历史记录上，而第三句则可以——例如，当我们把“现在”换成2000年9月25日时。对于历史数据来说，“从……到……”是个很重要的概念，特别是对于状态数据而言[⊖]。

术语：事件发生的时刻或保持某种状态的时间间隔均称为有效时间。命题p的有效时间是命题p为真的时间集合。它与事务时间不同，事务时间是数据库中某个命题为真的时间集合。有效时间可以随着命题的变化而变化，而事务时间则不能。也就是说，事务时间完全由系统管理，用户无法进行更改（事务时间一般被记录在事务日志中）。

注意：前文中提到的时间间隔和时间集合包含了开始时刻s和终止时刻e，其中的时间t满足$s \leq t \leq e$。尽管似乎是显而易见的，但它还有一些值得研究的性质，我们将在下面的小节中详细阐述。

现在，可能会对以上的讨论产生如下疑问，下面给出相应的解答。

1) “所有满足$s \leq t \leq e$的时间t”是否是一个无限元素的集合，那么在这个集合上是否无法进行计算呢？

   解答：不错，看上去它似乎是个无限元素的集合，而实际上我们把时间视为有限个时间单位的集合，因此上面的集合中只包含了有限个元素。

   注意：有人把时间单位称为chronon，但是这样一来又可以把chronon定义为一个时间间隔（参见[22.2]中的词汇表），它就具有起点和终点，这样又会无穷无尽地分割下去。为了避免误解，我们不采用这种记法。

2) 在例子中的第一、二和三句中似乎假设时间单位为天，但是很显然系统中应该支持更小的时间单位如秒。如果供应商S1在1999年7月1日签订合同，那么系统又该如何处理从1999年7月1日0时到他签订合同的时刻这一段时间呢？

   解答：我们需要正确区分系统所能表示的最小时间单位和出于某种目的而划分的时间单位（可能是年、月、日或星期等）。这种特定的时间单位称为时间点，不可再分。通

⊖ 尽管我们多次用到了“历史记录”等术语，但时态数据库中也包含与将来相关的信息。例如，供应商S1将会在日期a到b间签约，这里a和b都是将来的日期。

俗地讲，时间点是一个时间间隔（即时间单位的集合），从一个时间边界到另外一个时间边界（例如，从某日的午夜到第二天的午夜）。因此可以通俗地说：时间点是一个连续时间段——在本例中为一天。但按照正式的说法，时间点是不可分的，因此就不包含这种连续性。

注意：有些资料里用颗粒（granule）来代替时间点概念。从时间的观点上看，颗粒实际上就是区间，因此我们不采用颗粒的概念[⊖]。我们采用粒度(granularity)概念，定义为可用时间点的持续时间。因此，在本例中，粒度为天，这样就抛开了一天是由小时组成，小时又是由分钟组成的事实（这种事实可以用更细的粒度来描述）。

3) 既然时间是由一系列时间点（具有某种粒度）组成，那么我们是否可以明确地使用“前一个时间”或“后一个时间”呢？

解答：不错，后一个时间指的是沿时间终点方向的下一个时间点，前一个时间指的是沿时间起点方向的上一个时间点。时间的起点是一个没有前驱的时间点，时间的终点是一个没有后继的时间点。

4) 如果在关系中包含了事实“供应商S1的合同有效期为1999年7月1日到2000年9月25日”，那么它是否就不能包含事实“供应商S1的合同有效期为1999年7月2日到2000年9月24日”呢？

解答：问得好！很显然，需要给出如下的约束断言：“供应商S*x*签订的合同在日期*s*到*e*中的每一天均有效，但在*s*的前一天和*e*的后一天无效。”[⊜] 相应的细节请参见第22.8节和22.10节。

22.3 问题是什么

我们仍然以供应商-零件数据库为例，但需要根据需求进行适当修改。修改的步骤如下：首先为了简化问题删除零件关系变量P。然后删除发货关系变量SP中的QTY属性（只留下S#和P#），SP对应的语义是：“供应商S#目前可以供应零件P#”（换句话说，不再包含发货量信息，而只包含供应商现有零件的信息——即供应商现在能供应哪些零件）。图22-1是第3章图3-8的修改，给出了一些示例数据。注意这个数据库仍然只是个快照式数据库，没有包含任何时态信息。

S

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

SP

S#	P#
S1	P1
S1	P2
S1	P3
S1	P4
S1	P5
S1	P6
S2	P1
S2	P2
S3	P2
S4	P2
S4	P4
S4	P5

图22-1 供应商和发货数据示例(示例数据)——当前快照版

⊖ 时间单位和颗粒是否是区间，就像直觉和形式之间的差别一样容易让人混淆。从直觉上说的事情和它的形式可能会完全不同。尽管我们相信时间是连续的和无限的，但在计算机上我们却认为它是离散而有限的。

⊜ 本章中的“预测”与第8章中的外部或用户可理解的预测概念相同，而并不是内部或计算机可理解的预测。

接下来探讨这个数据库上的一些简单约束和查询。以后将考虑在数据库中加入时态数据时会对这些约束和查询有何影响。

约束（当前的快照式数据库）：我们要考虑的唯一约束是主码约束。{S#}和{S#，P#}分别是S和SP的主码，{S#}是SP的外码。

查询（当前的快照式数据库）：我们只考虑两个很简单的查询：

- 查询1.1：查询当前可以供应某种零件的供应商的数目。

```
SP {S#}
```

- 查询1.2：查询当前不能供应任何零件的供应商的数目。

```
S {S#} MINUS SP {S#}
```

查询1.1是个简单的投影操作，查询1.2则是两个此类投影的差。当我们考虑这两个查询的时态形式时，会发现对上述两个操作存在相应的时态形式（参见第22.8节）。注意：其它关系操作也可以定义相应的时态形式。

1. “半时态化”供应商和发货

下一步是分别对关系变量S和SP进行“半时态化”，添加时间戳属性，并改名。参见图22-2。

S_SINCE

S#	SNAME	STATUS	CITY	SINCE
S1	Smith	20	London	*d04*
S2	Jones	10	Paris	*d07*
S3	Blake	30	Paris	*d03*
S4	Clark	20	London	*d04*
S5	Adams	30	Athens	*d02*

SP_SINCE

S#	P#	SINCE
S1	P1	*d04*
S1	P2	*d05*
S1	P3	*d09*
S1	P4	*d05*
S1	P5	*d04*
S1	P6	*d06*
S2	P1	*d08*
S2	P2	*d09*
S3	P2	*d08*
S4	P2	*d06*
S4	P4	*d04*
S4	P5	*d05*

图22-2 供应商和发货数据库（示例数据）——半时态化版本

为了简化显示，在图22-2中不出现实际的时间戳数据，而用符号*d01*，*d02*等代替，这里可以把“*d*”读作“day”，本章中将使用这种命名约定（因此所有例子中都会以天作为时间点）。假设*d01*，*d02*，*d03*……是一个顺序序列，其中的无关紧要的0可以忽略，比如“day 1”。

S_SINCE的断言是“自从日期SINCE开始，供应商S#已被命名为SNAME，状态为STATUS，位于城市CITY，并已建立合同关系”。SP_SINCE的断言是“自从日期SINCE开始，供应商S#可以供应零件P#”。

约束（半时态化数据库）：这个“半时态化的”数据库中的主码和外码与原来数据库的相同。不过，需要一个附加约束——可视为对外码约束（从SP_SINCE到S_SINCE）的扩充——在没有和供应商建立合同关系之前，它不能供应任何零件。换句话说，如果SP_SINCE中的元组*sp*中引用了S_SINCE中的元组*s*，那么*sp*中的SINCE值必须不小于*s*中的SINCE值：

```
CONSTRAINTAUG_SP_TO_S_FK
   IS_EMPTY (((S_SINCE  RENAME  SINCE  AS ) JOIN
             (SP_SINCE  RENAME SINCE  AS  ) )PS
             WHERE  SPS<S)S;
```

从这个例子可以发现问题。假设“半时态化的”数据库如图22-2所示，那么我们就不得

不声明很多与上例类似的“附加外码约束”，这显然是我们希望避免的。

查询（半时态化数据库）：现在考虑查询1.1和查询1.2的“半时态化”版本。

• 查询2.1：查询当前可以供应某种零件的供应商的数目，并分别显示从何时起可以供应。

如果供应商S*x*现在可以供应一些零件，那么S*x*可以供应零件的日期就是SP_SINCE中S*x*对应的SINCE的最小值（例如，S1对应的最早SINCE日期为*d04*）。因此：

```
SUMMARIZE  SP  PER  SP  {S#}  ADD  MIN  (SINCE)  AS  SINCE
```

查询结果：

S#	SINCE
S1	*d04*
S2	*d08*
S3	*d08*
S4	*d04*

• 查询2.2：查询当前不能供应任何零件的供应商的数目，并分别显示从何时起无法供应。

在我们的示例数据中可以看到，现在只有一位供应商无法供应任何零件，就是供应商 *S5*。但是我们却无法得知它从何时起无法供应零件（而此时又已经和它建立了合同关系），因为数据库中缺少相关信息——还只是一个“半时态化的”数据库。例如，假设现在的日期为 *d10*，那么*S5*可能在日期*d02*到*d09*都可以供应零件，也可能从来就无法供应任何零件。

要回答查询2.2，就必须完全“时态化”整个数据库，或者至少要完全“时态化” SP。具体地说，就要在数据库中保存历史记录，反映哪个供应商在何时可以供应何种零件。

2. 完全时态化供应商和发货

图22-3显示了供应商和发货的完全时态化版本。其中将属性 SINCE改为FROM，并增加了时间戳属性TO。属性FROM和TO合起来表示一个时间间隔，在此时间间隔中某种命题为真。因为保存了历史记录，因此数据库中的元组数目会增多。假设现在的日期为 *d10*，那么每个属性TO的值为*d10*的元组中就包含了其当前状态。注意：大家可能会想到该采用什么机制来在午夜将所有的*d10*更新为*d11*，可惜现在我们不会去讨论它，请参见第 22.11节。

图22-3中的时态数据库包含了图22-2中的半时态数据库中的所有信息，并增加了以前的历史信息——在时间间隔（从 *d02*到*d04*）中与供应商 S2有合同关系。 S_FROM_TO的断言是“在时间间隔（从 FROM到TO）中，供应商S#被命名为SNAME，状态为STATUS，位于城市CITY，并建立合同关系”。SP_FROM_TO的断言是“在时间间隔（从 FROM到TO）中，供应商S#可以供应零件P#。”

约束（第一个时态数据库）：首先在FROM-TO中TO对应的时间点不能早于 FROM对应的时间点：

```
CONSTRAINT S_FROM_TO_OK
   IS_EMPTY (S_FROM_TO WHERE TO < FROM);
CONSTRAINT SP_FROM_TO_OK
   IS_EMPTY (SP_FROM_TO WHERE TO < FROM);
```

其次从图22-3中可以看出FROM属性是S_FROM_TO和SP_FROM_TO的主码属性（双下划线）；例如，S_FROM_TO的主码不能只是{S#}，否则无法保存在一段连续时间内与同一个供应商建立合同关系的信息。对 SP_FROM_TO也一样。注意：也可以用 TO代替FROM作为主码属性；实际上，在 S_FROM_TO和SP_FROM_TO中都有两个候选码，选取哪个作为主码

都可以[8.13]。

S_FROM_TO

S#	SNAME	STATUS	CITY	FROM	TO
S1	Smith	20	London	*d04*	*d10*
S2	Jones	10	Paris	*d07*	*d10*
S2	Jones	10	Paris	*d02*	*d04*
S3	Blake	30	Paris	*d03*	*d10*
S4	Clark	20	London	*d04*	*d10*
S5	Adams	30	Athens	*d02*	*d10*

SP_FROM_TO

S#	P#	FROM	TO
S1	P1	*d04*	*d10*
S1	P2	*d05*	*d10*
S1	P3	*d09*	*d10*
S1	P4	*d05*	*d10*
S1	P5	*d04*	*d10*
S1	P6	*d06*	*d10*
S2	P1	*d02*	*d04*
S2	P2	*d03*	*d03*
S2	P1	*d08*	*d10*
S2	P2	*d09*	*d10*
S3	P2	*d08*	*d10*
S4	P2	*d06*	*d09*
S4	P4	*d04*	*d08*
S4	P5	*d05*	*d10*

图22-3 供应商和发货数据库（示例数据）——第一个完全时态化版本

不过主码本身还需要进一步的约束。以关系变量SP_FROM_TO为例，很显然，如果存在元组（供应商为S*x*，FROM为*f*，TO为*t*），那么就不可能再出现元组（供应商为S*x*，FROM的值为*f*的前一天或者TO的值为*t*的后一天）。例如，在SP_FROM_TO中存在元组（*S1, d04, d10*），那么尽管存在主码约束{S#，FROM}，还是可能会出现元组（*S1*，*d02*，*d06*）这种重叠现象，它表明刚刚在*d04*之前与供应商*S1*建立了合同关系。很显然，我们希望将这两个元组合并为（*S1*，*d02*，*d10*）[⊖]。

另外，只有主码约束也不能阻止“邻接”元组的出现，如（*S1*，*d02*，*d03*），同样表明刚刚在*d04*之前与供应商*S1*建立了合同关系。和上面提到的一样，要将它们合并起来。

下面给出防止重叠和邻接出现的约束：

```
CONSTRAINT AUG_S_FROM_TO_PK
   IS_EMPTY (((S_FROM_TO RENAME FROM AS F1, TO AS T1) JOIN
                     (S_FROM_TO RENAME FROM AS F2, TO AS T2))
                     WHERE (T≥F2 AND T≥F1)) OR
                                 (F2=T1+1 OR F1=T2+1));
```

这个约束表达式相当复杂！——“T1+1”表示T1的直接后继，在第22.5节中会进行阐述。注意：当已经施加了这个约束之后，有些设计人员可能会把{S#，FROM，TO}作为时态候选码（时态主码），但这种做法并不好，因为“时态”候选码实际上未必是关系变量真正的候选码！（在第22.9节存在一个反例，其中的“时态候选码”就是传统意义上的候选码）。

接下来要注意关系变量SP_FROM_TO中的属性组合{S#，FROM}不是从SP_FROM_TO到S_FROM_TO的外码（尽管在S_FROM_TO的主码中也包含这些属性）。可是，如果在SP_FROM_TO中包含了某个供应商，那么它也应该包含于S_FROM_TO中：

```
CONSTRAINT AUG_SP_TO_S_FK_AGAIN1
```

⊖ 如果不进行元组的合并，则会产生与重复类似的副作用！重复指的是“两次描述相同的事物”。而具有重叠时间间隔的S1元组也是“两次描述相同的事物”；它们都描述了S1在日期4、5和6都签定了合同。

```
   SP_FROM_TO {S#} ⊇ S_FROM_TO {S#};
```

（“⊇”表示“包含于”）。

但是仅仅只有约束AUG_SP_TO_S_FK_AGAIN1依然是不够的；我们还要保证——即使已经进行了元组合并——如果SP_FROM_TO中显示了某个供应商在某个时间间隔可以供应某种零件，那么S_FROM_TO中就应该显示这个供应商在同一个时间间隔已经建立了合同关系。也许可以建立如下约束：

```
CONSTRAINT AUG_SP_TO_S_FK_AGAIN2   /* 警告——错误！ */
   IS_EMPTY ((S_FROM_TO RENAME FROM AS SF, TO AS ST) JOIN
                     (SP_FROM_TO RENAME FROM AS SPF, TO AS SPT))
                     WHERE SPF < SF OR SPT>ST);
```

正如注释里所示，这个约束说明是不正确的。例如，假设S_FROM_TO如图22-3所示，然后在SP_FROM_TO中添加元组（供应商为S2，FROM=*d03*，TO=*d04*），这样做显然是合理的，但是却被上面的约束所禁止。

我们会在第22.9节中解决这个问题，在这里只想指出，在关系变量SP_FROM_TO中采用{S#，FROM，TO}作为“时态候选码”以及在关系变量S_FROM_TO中采用{S#，FROM，TO}作为“时态外码”是不合适的。进一步的讨论请参见第22.9节。

查询（第一个时态数据库）：查询1.1和1.2的完全时态化版本：

- 查询3.1：查询在某个时间间隔可以供应某种零件的供应商元组S#-FROM-TO，其中FROM和TO合起来表示供应商S#可以供应某种零件的最长时间间隔。注意：“最长”的含义是指在此时间间隔开始的前一天和结束的后一天供应商S#无法供应任何零件。
- 查询3.2：查询在某个时间间隔内不能供应任何零件的供应商元组S#-FROM-TO，其中FROM和TO合起来表示供应商S#无法供应任何零件的最长时间间隔。

大家可能认为根本不会作这样的查询！但是这些查询还是可以表述出来，尽管需要多花些精力去实现。

总而言之，时态数据的问题在于它将导致一些很难表述的约束和查询——除非系统提供了优良设计的表示形式，当然在现有的商用DBMS产品中尚未出现。

22.4 时间间隔

我们现在就开始开发一些合适的表示符号。最初的，同时也是最基本的步骤是将时间间隔定义为一个基本单位，而不是像前文中作为一对分离值处理。

时间间隔到底是什么？根据图22-3，供应商S1可以在时间间隔（从日期4到日期10）中供应零件P1。但“从日期4到日期10”指的是什么呢？很显然，它包括日期5，6，7，8和9——那么起始终止日期（日期4和日期10）是否包含在内呢？有时我们希望时间间隔包含起始终止日期，而有时则希望它不包括起始终止日期。如果时间间隔包含开始日期4，那么我们说时间间隔对于起始日期是封闭的；否则说它对于起始日期是开放的。同样，如果时间间隔包含日期10，那么我们说时间间隔对于终止日期是封闭的；否则说它对于终止日期是开放的。

按照惯例，用括号把起始终止日期括起来表示时间间隔，圆括号表示时间间隔在此点上是开放的，方括号则表示时间间隔在此点上是封闭的；例如，从日期4到日期10的时间间隔有如下四种：

[d04, d10]
[d04, d10)
(d04, d10]
(d04, d10)

注意：可能认为半开半闭的时间间隔有点奇怪，不过这四种时间间隔各有实际意义，实际上常常会用到这种半开半闭的时间间隔。全封闭的时间间隔则很直观，我们会在下文中经常使用[⊖]。

如果用诸如[*d04*，*d10*]的形式来表示时间间隔，那么就可以将 SP_FROM_TO（参见图22-3）中的属性FROM和TO合并起来，用单个属性 DURING来表示，它的值域为 interval type（时间间隔类型，参见下一节）。这样做的一个直接好处是避免了从 {S#，FROM}和{S#，TO}中随意选择一个作为主码；另一个直接好处是不会对起始终止时间是否封闭产生歧义，因为可以用[*d04*，*d10*]、[*d04*，*d10*)、(*d04*，*d10*]、(*d04*，*d10*)四种形式来表示时间间隔。还有一个好处是不再需要时间间隔约束“ 在 FROM-TO中TO对应的时间点不能早于FROM对应的时间点 ”（参见第22.3节），因为已经在时间间隔类型中隐含地包括了约束“ FROM≤TO ”。第4个好处是不需要讨论哪些是非真正码的“ 时态码 ”（参见第22.9节），很多约束也可以得到简化（参见第22.9节）。

图22-4中显示了采用上述方法后数据库的变化。

S_DURING

S#	SNAME	STATUS	CITY	DURING
S1	Smith	20	London	[d04,d10]
S2	Jones	10	Paris	[d07,d10]
S2	Jones	10	Paris	[d02,d04]
S3	Blake	30	Paris	[d03,d10]
S4	Clark	20	London	[d04,d10]
S5	Adams	30	Athens	[d02,d10]

SP_DURING

S#	P#	DURING
S1	P1	[d04,d10]
S1	P2	[d05,d10]
S1	P3	[d09,d10]
S1	P4	[d05,d10]
S1	P5	[d04,d10]
S1	P6	[d06,d10]
S2	P1	[d02,d04]
S2	P2	[d03,d03]
S2	P1	[d08,d10]
S2	P2	[d09,d10]
S3	P2	[d08,d10]
S4	P2	[d06,d09]
S4	P4	[d04,d08]
S4	P5	[d05,d10]

图22-4 供应商和发货数据库（示例数据）——最终的完全时态化版本，使用时间间隔

22.5 间隔类型

上一节中只是对时间间隔做了直观的讨论，现在来将它抽象化。首先我们看到时间间隔[*d04*，*d10*]的粒度是“ 天 ”，更精确地说，它是 DATE（日期）类型，是 DATETIME（日期时间）类型的特例（以“ 天 ”为单位，而不是“ 小时 ”或“ 毫秒 ”或“ 月 ”）。可以对时间间隔值[*d04*，*d10*]的类型严格定义如下：

- 首先，它当然是一种时间间隔类型，从而决定了对其可以进行哪些操作（就像如果已知

[⊖] 采用半闭半开区间类型的优点：例如，将 [*d04, d10*]从*d07*分开，结果为两个相临区间 [*d04, d07*]和[*d07, d10*]。

*r*属于某种关系类型，就决定了对应的操作如 JOIN）。

- 其次，这个时间间隔值是从某一天到另一天，从而决定了它的值域。

因此，[*d04*，*d10*]所属的类型是INTERVAL(DATE)，其中：

a. INTERVAL是类型生成器（type generator，类似于Tutorial D中的关系 —— 参见第5章 —— 或者传统编程语言中的“数组”），用于定义各种时间间隔类型。

b. DATE是时间间隔类型中的点类型（point type）。

一般说来，对于 INTERVAL(PT)类型，点类型 PT定义了起点、终点和所有中间点的类型和精度（当点类型为DATE时，其精度隐式表示）。

注意：在第4章中我们说过，精度应该被视为一种完整性约束，而不是作为类型的一部分。假设给定如下声明DECLARE X TIMESTAMP(3)和DECLARE Y TIMESTAMP(6)，则X和Y属于相同类型，但满足不同约束（ X的取值单位为毫秒， Y的取值单位为微秒）。严格说来，TIMESTAMP(3) —— 或DATE —— 是同时包含了类型和精度的有效点类型，但最好分开处理。也许我们更希望定义两种类型 T1和T2，它们都可以表示时间戳类型，但具有不同“精度约束”，称T1和T2（而不是TIMESTAMP(3)和TIMESTAMP(6)）是有效点类型。为了便于理解，我们在本章中依然沿袭传统用法，将精度视为类型的一部分。

有效的点类型应该具有什么属性呢？我们看到间隔是用起点和终点来表示的，由一系列的点组成。如果已知点的全集，给定起点 *s*和终点*e*，就可以知道紧跟点 *s*的下一个点，称之为*s*的后继，简记为 *s*+1。根据*s*来判定*s*+1的函数称为点类型的后继函数。除了“ last ”点外，对于点类型中的每个取值都要定义其后继函数（“ first ”点不是任何点的后继）。

现在已知 *s*+1是*s*的后继，下面按照点类型中的顺序来判断 *s*+1是否在*e*之后。如果不是，那么*s*+1就是[*s*，*e*]中的点，接下来对下一个点 *s*+2做类似处理，直到*e*的后继*s*+*n*为止，这样就遍历了[*s*，*e*]中的所有点。

因为*s*+*n*是*e*的后继（即紧跟 *e*的下一个点），那么类型 *PT*是有效点类型的充要条件是具有后继函数，简而言之就是 *PT*中的点具有某种顺序关系（对于 *PT*中的每个点都定义了比较操作符 —— 如“ < ”、“ ≥ ”，等）。

大家一定注意到我们并没有将讨论限定在时态数据上。实际上，本章大部分内容都是讨论一般的间隔而不是时间间隔，在第 22.11节会讨论有关时间间隔的问题。

下面是更准确的定义：

- 假设*PT*是一种点类型。那么类型为 INTERVAL（*PT*）的间隔（或间隔值）*i*就是一个数量值，定义了两个一元操作符（START和END）和一个二元操作符（IN）如下：

a. START（*i*）和END（*i*）的返回值类型为*PT*；

b. START（*i*）≤END（*i*）；

c. 令*p*为*PT*类型的值，那么 *p* IN *i*为真的充要条件是 START（*i*）≤*p*，而且*p*≤END（*i*）；

定义中给出了类型 PT上的后继函数；起点和终点合起来表达了 INTERVAL（*PT*）的所有值。根据定义，间隔总是非空的（即在任意间隔中必然包含一个点）。

要注意类型 INTERVAL（*PT*）中值是标量值 —— 即它没有可视的部件。不错，每个值代表了一定的含义，它们合起来就成为可视的，但单个值并不是可视的。我们称间隔是封装的。

22.6 间隔上的标量操作符

本节中将会定义一些间隔值上的标量操作符。对于间隔类型 INTERVAL(*PT*)，令*p*为*PT*类型的一个值，可以使用标记 *p*+1，*p*+2等来表示*p*的后继，*p*+1的后继等（实际语言中可能会提供NEXT操作符）。同样地，可以使用标记 *p*-1，*p*-2等来表示*p*的前驱，*p*-1的前驱等（实际语言中可能会提供PRIOR操作符）。

令*p1*和*p2*为*PT*中的值，定义 MAX(*P1*,*P2*)为：当*p1*<*p2*时返回*p2*，否则返回*p1*。定义 MIN(*P1*,*P2*)为：当*p1*<*p2*时返回*p1*，否则返回*p2*。

以上定义可用于间隔选择（selector）。例如，[3,5]和[3,6]中的选择调用将返回包含了3、4和5这些点的INTERVAL（INTEGER）类型的值（实际语言中可能会要求更明确的语法，例如 INTERVAL（[3, 5]））。

令*i1*为间隔[*s1*，*e1*]，类型为INTERVAL（*PT*），则START(*i1*)返回*s1*，END(*i1*)返回*e1*，STOP(*i1*)返回*e1*+1。令*i1*为间隔[*s1*，*e1*]，类型为INTERVAL（*PT*），定义以下比较操作符。注意：这些操作符有时被称为 Allen操作符，因为它们最早是 Allen提出的，参见参考文献[22.1]。

- *i1*=*i2*为真，当且仅当*s1*=*s2*和*e1*=*e2*均为真。
- *i1* BEFORE *i2*为真，当且仅当*e1*<*s2*为真。
- *i1* MEETS *i2*为真，当且仅当*s2*=*e1*+1为真或*s1*=*e2*+1为真。
- *i1* OVERLAPS *i2*为真，当且仅当*s1*≤*e2*和s2≤*e1*均为真。
- *i1* DURING *i2*为真，当且仅当*s2*≤*s1*和*e2*≥*e1*均为真[㊀]。
- *i1* STARTS *i2*为真，当且仅当*s1*=*s2*和*e1*≤*e2*均为真。
- *i1* FINISHES *i2*为真，当且仅当*e1*=*e2*和*s1*≥*s2*均为真。

注意：同样可以根据点来定义这些操作符。例如，定义 *i1* OVERLAPS *i2*为真，当且仅当存在*PT*类型的值*p*，使得*p* IN *i1*和*p* IN *i2*均为真。

根据参考文献[22.3]，可以扩展Allen操作符的定义：

- *i1* MERGES *i2*为真，当且仅当*i1* MEETS *i2*为真或*i1* OVERLAPS *i2*为真。
- *i1* CONTAINS *i2*为真，当且仅当*i2* DURING *i1*为真[㊁]。

间隔长度用DURATION(*i*)定义，它返回*i*中点的个数。例如，DURATION([*d03*,*d07*])=5。

最后定义一些有用的二元操作符，它们返回一个间隔：

- *i1* UNION *i2*，若*i1* MERGES *i2*为真返回[MIN(*s1*,*s2*), MAX(*e1*,*e2*)]，否则无定义。
- *i1* INTERSECT *i2*，若*i1* OVERLAPS *i2*为真，返回[MAX(*s1*,*s2*),MIN(*e1*,*e2*)]，否则无定义。

注意：这里的UNION和INTERSECT就是一般的集合操作符。参考文献 [22.3]中用MERGE和INTERVSECT来标记这些操作符。

22.7 间隔上的聚集操作符

本节给出两个十分重要的操作符， UNFOLD和COALESCE。它们都是以一组相同类型的

㊀ 注意此处的DURING并不表示“问题中的整个区间”。

㊁ INCLUDE可能比CONTAINS更合适；可以用CONTAINS作为IN的逆，定义*i* CONTAINS *p*等同于*p* IN *i*。

间隔集合为参数，并返回相应的集合。可以将结果视为原始集合的一种范式（参见第 17章17.3节）。

令*X1*和*X2*为以下两个集合：

```
{ [d01,d01], [d03,d05], [d04,d06] }
```

和

```
{ [d01,d01], [d03,d04], [d05,d05], [d05,d06] }
```

很显然这两个集合是不同的，同时我们也可以轻易发现 (a)*X1*中所有间隔中所包含的点的集合等同于(b) *X2*中所有间隔中所包含的点的集合（这些点是 *d01*, *d03*, *d04*, *d05*和*d06*）。不过我们关心的不是这些点的集合，而是单位间隔（unit interval）的集合，记为*X3*：

```
{ [d01,d01], [d03,d03], [d04,d04], [d05,d05], [d06,d06] }
```

我们称*X3*为*X1*（和*X2*）的展开形式（unfolded form）。一般来说，如果*X*是一组相同类型的间隔集合，那么*X*的展开形式就是包含了所有形式为 [*p*,*p*]的间隔集合，其中*p*是*X*中某个间隔中的点。

注意*X1*、*X2*和*X3*的基数不同。上例中*X3*的基数最大，不过我们可以找到一个集合 *X4*，它和*X1*具有相同的展开形式，但其基数比 *X3*大（请读者作为练习）。同样可以找到唯一的一个集合*X5*，它和*X1*具有相同的展开形式，但基数最小：

```
{ [d01, d01], [d03,d06] }
```

我们称*X5*为*X1*的合并形式（coalesced form），同样它也是*X2*，*X3*和*X4*的合并形式。一般说来，如果*X*是一组相同类型的间隔集合，则 *X*的合并形式*Y*是同样类型的间隔集合，满足 (a)*X*和*Y*的展开形式相同，(b)在*Y*中不存在两个不同的成员 *i1*和*i2*，使得*i2* MERGES *i2*为真。注意多个不同集合可以具有相同的合并形式。另外要注意合并形式的定义与点类型上的后继函数相关，而展开形式则没有这种依赖关系。

现在可以定义操作符UNFOLD和COALESCE。令*X*为INTERVAL(*PT*)类型的间隔集合，则UNFOLD(*X*)返回*X*的展开形式，COALESCE(*X*)返回*X*的合并形式。注意：我们必须指出展开形式和合并形式并不是正规术语，实际上它们也没有对应的正规术语。

对于第22.3节中讨论的问题，这两个范式十分重要。不过 UNFOLD和COALESCE操作符还不是我们真正需要的（而只是其中的一环），我们真正需要的是这些操作符在关系模式中的对应概念，在下一节中将予以讨论。

22.8 与间隔有关的关系操作符

在关系表达式中当然可以使用第 22.6节中描述的标量操作符。例如，在 Tutorial D中，一般会在限制中的 WHERE子句或EXTEND和SUMMARIZE中的ADD子句中使用这些操作符。以图22-4中的数据库为例，查询“能够在日期 8供应零件p2的供应商个数”可以如下表达：

```
( SP_DURING WHERE P# = P# ('P2') AND DURING ) { S# }
```

注意：表达式中的*d08*可以是类型为DAY的任意值。

再看另一个例子，下面表达式的返回结果是同时位于同一个城市的供应商，以及相应的城市和时间：

```
EXTEND
   ( ( ( ( S_DURING RENAME S# AS XS#,
```

```
                        DURING AS XD ) { XS#, CITY, XD }
    JOIN
    (S_DURING RENAME S# AS YS#,
                  DURING AS YD ) { YS#, CITY, YD } )
    WHERE XD OVERLAPS YD )
AND ( XD INTERSECT YD ) AS DURING ) { XS#, YS#, CITY, DURING }
```

注解：使用JOIN来检索位于同一个城市的供应商。WHERE子句中限制了它们必须同时位于同一个城市。使用EXTEND...ADD来计算相关的间隔。最终的投影操作给出所需查询结果。

现在回头来看第22.3节末尾的查询3.1和查询3.2。查询4.1是查询3.1在图22-4中的数据库上的应用：

- 查询4.1：查询在某个时间间隔可以供应某种零件的供应商元组 S#-DURING，其中DURING表示供应商S#可以供应某种零件的最长时间间隔。

回忆一下此查询的更早版本，查询2.1，其中要用到分组和聚集操作（涉及SUMMARIZE操作）。无庸置疑，查询4.1中实际上也需要进行分组和聚集操作。不过，我们可以通过简单的几个步骤来构造这个查询。第一步是：

```
WITH SP_DURING { S#, DURING } AS T1 :
```

这一步中只是缺少零件号。T1如下所示：

S#	DURING
S1	[d04,d10]
S1	[d05,d10]
S1	[d09,d10]
S1	[d06,d10]
S2	[d02,d04]
S2	[d03,d03]
S2	[d08,d10]
S2	[d09,d10]
S3	[d08,d10]
S4	[d06,d09]
S4	[d04,d08]
S4	[d05,d10]

注意这个关系中包含一些冗余信息：比如其中至少包含了三条关于供应商S1在日期6可以供应某些零件的相同信息。去除这些冗余后的结果如下所示（称为结果）：

S#	DURING
S1	[d04,d10]
S2	[d02,d04]
S2	[d08,d10]
S3	[d08,d10]
S4	[d04,d10]

称这个结果为T1在DURING中的合并形式。必须注意的是在合并形式中，对于给定供应商的DURING值不一定会显式地出现在T1中。在本例中，对于S4就是如此。

最终可以通过如下表达式得到上面的合并形式：

```
T1 COALESCE DURING
```

不过需要逐步到达这个目标。

在前两段中提到的“合并形式”与第22.7节中的概念有细微的区别。第22.7节中的COALESCE操作符的输入参数是间隔集合，输出结果也是间隔集合。而在本节中讨论的COALESCE操作符可视为第22.7节的重载（overloading），它的输入参数是一元关系，输出参

数也是（具有相同标题的）一元关系，关系中的元组包含了实际的间隔。

下一步是从T1中得到结果：

```
WITH ( T1 GROUP ( DURING ) AS X ) AS T2 :
```

（GROUP操作符参见第6章）。T2如下所示：

S#	X
S1	DURING [d04,d10] [d05,d10] [d09,d10] [d06,d10]
S2	DURING [d02,d04] [d03,d03] [d08,d10] [d09,d10]
S3	DURING [d08,d10]
S4	DURING [d06,d09] [d04,d08] [d05,d10]

现在将重载后的COALESCE操作符应用在关系值属性X上：

```
WITH ( EXTEND T2 ADD COALESCE ( X ) AS Y )
                              { ALL BUT X } AS T3 :
```

T3如下所示：

S#	Y
S1	DURING [d04,d10]
S2	DURING [d02,d04] [d08,d10]
S3	DURING [d08,d10]
S4	DURING [d04,d10]

最后，撤消分组（参见第6章）：

```
T3 UNGROUP Y
```

上面的表达式产生所需的结果。我们现在将各个步骤合并起来形成一个综合表达式：

```
WITH SP_DURING { S#, DURING } AS T1,
    ( T1 GROUP ( DURING ) AS X ) AS T2,
    ( EXTEND T2 ADD COALESCE ( X ) AS Y ) { ALL BUT X } AS T3 :
T3 UNGROUP Y
```

显然，我们希望通过一次操作就可以从 T1得到结果，于是发明了新操作符“ 关系合并 ”，其语法如下：

```
R COALESCEA
```

（其中*R*是关系表达式，*A*是某种间隔类型的属性 —— 表达式所代表关系的属性）[⊖] 。此操作符的语义是分组、扩展、投影和逆分组操作的归纳。注意： *R*在*A*上的合并涉及到*R*在全部属性（*A*除外）上的分组（在第 6章中提到，表达式“ T1 GROUP (DURING)... ”可读作“ T1按S#分组 ”，S#是除了GROUP子句中提到的属性之外的所有其他属性）。

综上所述，查询4.1可如下表示：

```
SP_DURING { S#, DURING } COALESCE DURING
```

有些学者称之为时态投影。具体地说，它是 SP_DURING在S#和DURING上的投影。（这个查询的原始版本，查询 1.1，涉及SP在S#上的投影）。时态投影并非真正的投影操作，而是投影操作的时态模拟。

现在来看查询3.2，查询4.2是根据图22-4中的数据库对查询3.2的重新描述：

- 查询4.2：查询在某个时间间隔内不能供应任何零件的供应商元组 S#-DURING，其中DURING表示供应商S#无法供应任何零件的最长时间间隔。

此查询的原始版本，查询 1.2，涉及关系求差操作。“ 时态投影 ”需要“ 关系合并 ”，而“ 时态求差 ”则需要“ 关系展开 ”。

“ 时态求差 ”涉及两个关系操作数。首先来看左边的操作数。如果展开了 S_DURING { S#,DURING }在DURING上的投影结果，将得到如下关系 —— 称为T1：

S#	DURING
S1	[d04,d04]
S1	[d05,d05]
S1	[d06,d06]
S1	[d07,d07]
S1	[d08,d08]
S1	[d09,d09]
S1	[d10,d10]
S2	[d07,d07]
S2	[d08,d08]
S2	[d09,d09]
S2	[d10,d10]
S2	[d02,d02]
S2	[d03,d03]
S2	[d04,d04]
S3	[d03,d03]
..	.....

对于图22-4中的示例数据，T1中共有23个元组（练习：请检验）。

如果定义了“ 一元关系 ”上的 UNFOLD（类似于“ 一元关系 ”上的 COALESCE），将得到如下的T1：

⊖ *A*可以是用逗号分开的一组属性。

```
( EXTEND ( S_DURFING { S#, DURING } GROUP ( DURING ) AS X )
ADD UNFOLD ( X ) AS Y ) { ALL BUT X } UNGROUP Y
```

使用新的“关系扩展”操作符来简写上面的语句：

```
R UNFOLD A
```

现在就可以如下书写

```
WITH ( S_DURING { S#, DURING } UNFOLD DURING ) AS T1:
```

右边的“时态求差”操作数可被视为：

```
WITH ( SP_DURING { S#, DURING } UNFOLD DURING ) AS T2:
```

现在可以应用关系求差：

```
WITH ( T1 MINUS T2 ) AS T3 :
```

T3如下所示：

S#	DURING
S2	[d07,d07]
S3	[d03,d03]
S3	[d04,d04]
S3	[d05,d05]
S3	[d06,d06]
S3	[d07,d07]
S5	[d02,d02]
S5	[d03,d03]
S5	[d04,d04]
S5	[d05,d05]
S5	[d06,d06]
S5	[d07,d07]
S5	[d08,d08]
S5	[d09,d09]
S5	[d10,d10]

最后，在DURING上合并T3，得到如下结果：

```
T3 COALESCE DURING
```

结果如下所示：

S#	DURING
S2	[d07,d07]
S3	[d03,d07]
S5	[d02,d10]

查询4.2可以用单个嵌套表达式描述：

```
( ( S_DURING { S#, DURING } UNFOLD DURING )
MINUS
( SP_DURING { S#, DURING } UNFOLD DURING ) )
COALESCE DURING
```

这个综合操作是时态差（temporal difference）的一个例子。更准确地说，它是用S#和DURING上的S_DURING投影减去SP_DURING投影所得到的差。注意：和时态投影一样，时态差并不是真正的差，而是我们通常所说的差的“时态模拟”。

不过事情还没有结束。由于常常要用到上例中出现的“时态差”表达式，所以要定义一个简写形式，它包含一系列操作(a)合并两个操作数；(b)差的计算；和(c)合并 ⊖。简写形式还

⊖ 注意我们并没有为时态投影定义相应的简写形式。

可以提高性能。当涉及较细粒度的长间隔时，对于操作数来说，关系合并的结果会很庞大；如果在系统中实例化展开形式和差，然后再合并结果，那么查询就可能陷入死循环或造成磁盘溢出。将临时求差作为单步操作有助于优化查询处理，避免不必要的展开。简写形式如下：

```
R1 I_MINUS R2 ON A
```

其中，*R1*和*R2*是关系表达式，表示相同类型的关系 *r1*和*r2*，*A*是这两个关系共有的某间隔类型的一个属性（前缀“ I- ”表示“ 间隔 ”）。如我们所已知的，该表达式定义为在语义上等价于以下内容：

```
( ( R1 UNFOLD A ) MINUS ( R2 UNFOLD A ) ) COALESCE A
```

练习22.2对“ I- ”操作符，如I-MINUS有进一步的讨论。

22.9 间隔上的约束

很显然对于变量 S_DURING来说，复合属性 [S#,DURING]是一个候选码；在图 22-4中，我们使用双下划线来标明主码（ {S#}不能作为候选码，因为某个供应商可以在合同结束后重新开始新的合同 —— 例如图22-4中的供应商S2）。因此，变量S_DURING可以如下定义：

```
VAR S_DURING BASE RELATION
{ S# S#, SNAME NAME, STATUS INTEGER, CITY CHAR,
DURING INTERVAL ( DATE ) }
KEY { S#, DURING } ;  /* 警告 —— 不充分 */
```

但是这里指定的主码仍然是不充分的，因为它无法禁止在变量 S_DURING中同时出现如下两个元组：

S2	Jones	10	Paris	[d02,d06]
S2	Jones	10	Paris	[d07,d10]

正如大家所看到的，这两个元组包含了一定的冗余，关于供应商 S2在日期7和8的信息被记录了两次。

码的不充分性还表现在其它方面，它无法禁止在变量S_DURING中同时出现如下两个元组：

S2	Jones	10	Paris	[d02,d06]
S2	Jones	10	Paris	[d07,d10]

这里虽然不再有冗余，但是存在一种迂回，实际上我们可以把这两个元组合并为一个：

S2	Jones	10	Paris	[d02,d10]

为了避免出现这种冗余或迂回，应该在下面的语句中强制一种变量约束 —— 称为约束C1：

如果两个S_DURING元组可以在DURING值（分别为*i1*和*i2*）上区分，那么*i1* MERGE *i2*就应该为假。

(因为MERGES是OVERLAPS或MEETS，如果在约束C1中用OVERLAPS来代替MERGES，则可以避免冗余；如果用 MEETS来代替MERGES，则可以避免迂回)。可以使用一种很简单的方法来强制 C1约束：让变量 S_DURING在属性DURING上总是合并的。下面定义一个新的COALESCED子句：

```
VAR S_DURING BASE RELATION
{ S# S#, SNAME NAME, STATUS INTEGER, CITY CHAR,
```

```
    DURING INTERVAL ( DATE ) }
    KEY { S#, DURING }     /* 警告 —— 还不充分 */
    COALESCED DURING ;
```

这里COALESCED DURING的含义是对于表达式S_DURING COALESCE DURING的结果，关系变量S_DURING必须总是唯一的（因此，S_DURING无须在DURING上进行合并）。上面的语法定义解决了冗余和迂回的问题[⊖]。注意：假定对于S_DURING的任何更新，如果它使得S_DURING在DURING上不是一种完全的合并形式，则禁止这个更新。参见第22.10节的进一步讨论。

但是仅仅指定了KEY和COALESCED仍然不够，在S_DURING关系变量中还是可能会包含如下元组：

S2	Jones	10	Paris	[*d02,d08*]
S2	Jones	20	Paris	[*d07,d10*]

在上表中可以看到，供应商S2在日期7和8同时具有两种状态10和20 —— 这显然是不可能的。也就是说，出现了一种矛盾情况。

为了避免这种矛盾，就需要进一步对关系变量进行约束 —— 称之为约束C2 —— 如下所示：

> 如果两个不同的S_DURING元组具有相同的S#值，对应的DURING值为*i1*和*i2*，并且*i1* OVERLAPS *i2*为真，那么这两个元组的其他属性（DURING除外）值应该是相同的。

值得注意的是，为了保证约束C2，并不要求在DURING属性上对S_DURING进行合并（因此{S#,DURING}未必是候选码），而是假定关系变量S_DURING在DURING属性上始终是展开的。那么：

- 对于展开形式S_DURING UNFOLD DURING来说，唯一的候选码应该是{S#,DURING}（因为在任何时间，每个建立合同关系的供应商只能有一个名字、一个状态和一个所在城市）。
- 因此，不会出现具有相同S#值和“重叠”DURING值的两个不同元组（因为在S_DURING UNFOLD DURING中的所有DURING值都是单位间隔，具有相同S#值和“重叠”DURING值的两个元组实际上就是同一个元组）。

这也就是说，如果限定{S#,DURING}为S_DURING UNFOLD DURING上的候选码，约束C2也“自动地”被保证。在关系变量定义KEY子句中增加如下的I_KEY子句(“I_”表示间隔)定义：

```
VAR S_DURING BASE RELATION
   { S# S#, SNAME NAME, STATUS INTEGER, CITY CHAR,
     DURING INTERVAL ( DATE ) }
   I_KEY { S#, DURING UNFOLDED }
   COALESCED DURING ;
```

（准确地说，{S#, DURING}是S_DURING UNFOLD DURING上的候选码）[⊜]。使用这个特定语法可以避免出现矛盾情况。

值得注意的是，如果{S#, DURING}是S_DURING UNFOLD DURING上的候选码，它也肯定是S_DURING的候选码；我们可以删除S_DURING中的KEY约束，用I_KEY约束来代替。另外{S#, DURING}还可被视为时态候选码（参见第22.3节）。我们可以看到，这个时态候选码实际上是关系变量的真正候选码（与第22.3节中讨论的时态候选码不同）。

⊖ 还应该讨论类似的语法定义，只用于解决冗余问题。

⊜ 某些作者类似定义I_KEY以解决冗余问题。这样做不太合逻辑，而且也不必要，因为COALESCED已经足以解决冗余问题。

当然，如果“ I_KEY ”语法可用于定义候选码，那么也可以用于定义外码。例如，SP_DURING的定义中可能会包括：

```
FOREIGN I_KEY { S#, DURING UNFOLDED } REFERENCES S_DURING ...
```

它的含义是，如果SP_DURING中显示了供应商S*x*在时间间隔*i*中可以供应某个零件，那么在S_DURING中就必须显示在时间间隔*i*中已经与供应商S*x*建立了合同关系。当满足了这个约束时，就可以把SP_DURING关系变量中的{S#, DURING}属性视为时态外码（同样它也并非传统意义上的外码）。

关于关系变量S_DURING，还有一点应该考虑。假设关系变量在DURING上始终处于合并状态，而且我们运行了一个程序来不断计算签约供应商的状态，程序显然要在 S_DURING中保存以前的状态值。有时这种计算的结果不会更改状态，程序就会在 S_DURING中插入一条保存以前状态的记录，这将会和COALESCED约束发生冲突！为了避免冲突，程序中应该测试“ 状态是否改变 ”。然后当状态不变时用合适的UPDATE来代替INSERT（参见本章末尾的练习22.3）。另外的一种解决方法是根本不在DURING上合并 —— 这种方法仅适合于某些情况。

22.10 间隔上的更新操作符

本节中讨论在时态关系变量上使用更新操作符 INSERT、UPDATE和DELETE所出现的一些问题。仍以S_DURING为例，假设定义中包含了上节中的 I_KEY和COALESCED约束，而且S_DURING中的当前值如图22-4所示。现在考虑下面情况：

- INSERT：假设我们发现在日期5和6中与供应商S2也具有合同关系（其他属性不变）。但却无法简单地插入一个元组来表述这个事实，因为那样会和 COALESCED约束冲突。要首先删除一个现有的 S2元组，然后更新另外一个 S2元组，将 DURING值设定为[*d02*,*d10*]。
- UPDATE：假设我们发现S2在日期9的状态临时地增加到20。尽管看起来只是一个简单的UPDATE，但实际上很难实现这种修改。需要将S2的[*d07*,*d10*]元组分割成三个元组，它们的DURING值分别是[*d07*,*d08*]、[*d09*,*d09*]和[*d10*,*d10*]，然后将[*d09*,*d09*]元组的状态修改为20。
- DELETE：假设我们发现S3的合同在日期6终止，然后在日期9重新签约。同样要将原来的单个元组分割为两个，对应的DURING值分别是[*d03*,*d05*]和[*d09*,*d10*]。

我们观察到这三种解决方法都和关系变量 S_DURING的当前值有关（也和要进行的修改有关）！例如，看一下插入问题，可能只是简单地插入新元组，也可能要和“ 前驱 ”或“ 后继 ”的元组合并。类似地，更新和删除操作就可能要“ 分割 ”现有的元组。

很显然，如果采用传统的INSERT、UPDATE和DELETE操作符，更新操作会很复杂，于是需要对这些操作符进行扩展。如下所示：

- INSERT：INSERT问题可以通过简单地扩展关系元组中的 COALESCED约束予以解决。首先执行基本的 INSERT操作，然后由系统执行所需的合并操作。换句话说，COALESCED不再仅仅是一个约束，同时蕴涵某种补充操作（对于外码同样处理）。然而COALESCED的语义扩展还无法解决UPDATE和DELETE问题。
- UPDATE：UPDATE问题可以通过对UPDATE操作符进行如下扩展解决[⊖] ：

⊖ 这种方法和参考文献[22.3]类似，但不相同。

```
UPDATE S_DURING
       WHERE S# =S# ('S2')
       DURING INTERVAL ([d09,d09])
       STATUS :=20;
```

第三行——它的语法基本上为<属性名><间隔表达式>——指定了合并约束的间隔属性（本例中为DURING）以及相应的间隔值（本例中为[*d09,d09*]）。整个UPDATE操作可以划分为如下步骤：

a. 首先标识出供应商S2对应的元组。

b. 然后在这些元组中标识出DURING属性值中包含[*d09, d09*]的元组（最多一个）。

c. 如果找不到满足条件的元组，就不进行任何更新；否则，切割元组，并做相应更新。

- DELETE：DELETE问题可以通过对DELETE操作符作类似扩展解决。例如：

```
DELETE S_DURING
       WHERE S#=S#('S3')
       DURING INTERVAL ([d06,d08]);
```

22.11 关于数据库设计

在关系变量S_DURING和SP_DURING中已经定义了间隔类型和相应的操作符，简单地增加了间隔属性。在本节中，我们讨论这种设计方法是否有效。特别地，我们引入某些关系变量上进一步的分解操作（超出规范化中所需的分解）。实际上，我们推荐在适当的情况下进行水平分解和垂直分解。

1. 水平分解

在运行实例中，我们可以合理地假设数据库中包含了到目前为止的历史数据；而且假设当前时间用特定日期记录（即日期10），但这种假设并不合理。特别地，随着时间的过去，数据库就必须相应更新（在实例中，在日期10的午夜，就必须把所有的日期10更改为日期11）。如果间隔的粒度更细，这种更新会更加频繁，甚至可能会每毫秒一次。

有些专著中提倡使用一个特殊标记——称之为now——来指定当前时间。使用这种方法，图22-4在S_DURING中供应商S1的DURING值——间隔[*d04,d10*]将用[*d04,now*]表示。间隔的实际值与用户观察数据的时间有关，例如，在日期14，这个间隔的值为[*d04,d14*]。

其他专著中认为在关系数据库中采用上面的方法是不严谨的。因为now只是一个变量，在值的记号中包含变量使得它在逻辑上混乱，如下所示：

- 在日期14的午夜，间隔[*now,d14*]该如何变化？
- 在日期14，END([*d04,now*])的值是什么？——是日期14还是*now*呢？

使用上面的方法显然无法给出一致的结果，因此需要更完善的解决方法。

有时“DURING属性”不仅记录过去的时间，还可能记录将来的时间。例如，可能需要记录供应商将来合同终止的日期或准备续约的日期。这种情况下可以继续使用图22-4中的S_DURING设计，但是，如果在DURING中包含了事务时间（参见第22.2节），则不能采用这种设计——因为事务时间中不能包含将来。

问题在于，在历史信息和涉及当前状态的信息之间存在重要区别：对于历史信息来说，起始和终止时间是已知的；而对于当前信息来说，起始时间已知，终止时间却很可能是未知的。由于存在这种区别，所以需要采用两个不同的关系变量，一个保存历史信息，另一个保

存当前信息（当然也就需要两个不同的谓词）。以供应商为例，“当前”关系变量是图 22-2中的S_SINCE，“历史”变量是图 22-4中的S_DURING（DURING值中终止时间为 *d10*的元组会移到S_SINCE中）。

本例中就用到了水平分解：使用一个关系变量来处理涉及当前状态的“ since”属性值，用另一个关系变量来处理涉及历史信息的“ during”属性值。前面我们提到了使用触发过程来处理历史关系变量，例如，从 S_SINCE中删除一条元组会“自动地”在 S_DURING中插入一条元组。

关系操作符UNION可用来将历史数据和当前数据合并为单个关系——例如：

```
S_DURING UNION ( EXTEND S_SINCE
                 ADD INTERVAL [ SINCE, TODAY() ]
                 AS DURING ) { ALL BUT SINCE }
```

如果DURING表示的是有效时间而非事务时间，水平分解可能会产生问题。此时历史数据是可以更改的！我们可以使用第 20.10节中提到的更新操作符，但是有的修订会同时影响两个关系变量。例如，假设我们发现最近错误地更改了某个供应商状态，那么不仅要从 S_DURING中删除相应元组，还要更新 S_SINCE中的相应元组。再如最近正确地更改了状态，但日期是错的，也需要同时更新两个关系变量。

如果也把 SP_DURING分解为 SP_SINCE和SP_DURING，则要考虑外码约束。对于 SP_DURING，其关系变量的定义中要包括（参见第 22.9节）：

```
FOREIGN I_KEY { S#, DURING UNFOLDED } REFERENCES S_DURING ...
```

正如第22.9节中提到的，如果供应商 S*x*在间隔*i*中可以供应某种零件，那么在 S_DURING中就要显示S*x*在间隔*i*中已经签订了合同。我们可以把 SP_DURING中的{S#,DURING}视为时态外码。

SP_SINCE中相应的外码只是“半时态化的”；因此必须增加如下约束（参见第 22.3节）：

```
CONSTRAINT AUG_SP_TO_S_FK
   IS_EMPTY ( ( S_SINCE RENAME SINCE AS SS ) JOIN
              ( SP_SINCE RENAME SINCE AS SPS ) )
         WHERE SPS < SS ) ;
```

因此，水平分解可能会导致某些问题——繁杂的约束，需要“同步”更新当前关系变量和历史关系变量。目前还没有什么简便的方法来解决这些问题，也许还需要进一步的研究。我们发现，如果在“ DURING”关系变量中除了包含过去和当前的信息外，还允许它包含将来的信息，则这些问题将不复存在（因为“ SINCE”关系变量可被删除）；但是这需要能预测将来的终止时间。相关讨论参见参考文献 [22.4]。

2. 垂直分解

在研究时态数据之前，甚至在 SQL被发明之前，人们就已经对关系变量的分解进行了充分研究，而不是仅限于传统的规范化。但是他们大多局限于二元关系变量，没有考虑一元关系变量和多元关系变量（例如供应商-零件-工程数据库中的 SPJ关系变量）。

（非时态化的）关系变量S显然可被进一步分解。假设“ S1的姓名是Smith”，“ S1的状态为 20”，“ S1位于伦敦”，我们可以推断出这些事实蕴涵在图 22-1中S的第一条元组之中，因此可以把S分解为三个二元关系变量，每个都以 S#作为主码。

这种分解是为了得到最简化的形式。对于关系变量 S来说可能无此必要，但对于

S_DURING来说却很必要。供应商的姓名、状态和所在城市与时间的相关度不同，更新的频率也不同。例如，基本不会更改供应商的姓名，偶尔会更改其所在城市，但却常常会更改它的状态——每次更改状态时重复姓名和所在城市是非常麻烦的。另外，考虑姓名、状态和城市各自的变化比姓名-状态-城市整个的变化更有意义。因此，将 S_DURING垂直分解为三个关系变量：

```
S_NAME_DURING     { S#, SNAME, DURING }
S_STATUS_DURING   { S#, SNAME, DURING }
S_CITY_DURING     { S#, CITY, DURING }
```

每个关系变量都要指定I_KEY {S#, DURING UNFOLDED }和COALESCED DURING。注意：可能还要包含“主”供应商关系变量：

```
S#_DURING { S#, DURING }
```

这个关系变量中指出了哪些供应商在何时已经签订了合同。同样也要指定 I_KEY {S#, DURING UNFOLDED }和COALESCED DURING。另外{ S#, DURING }可以作为S_NAME_DURING、S_STATUS_DURING和S_CITY_DURING的外码，与S_DURING中的时态候选码{ S#, DURING }对应。

此外，使用S_DURING的原始定义，我们还必须采用以下表达式来获取状态历史信息：

```
S_DURING { S#, STATUS, DURING } COALESCE DURING
```

这个表达式中仅仅用一个简单的关系变量引用来获取历史信息！因此，对查询做“运动场分级”分解可以简便地表达更有意义的信息，同时也使得对无关紧要信息的表达更加困难。

对于S_SINCE则不一定要进行同样的分解。特别要注意触发过程会同时更新三个历史关系变量——例如，从 S_SINCE中删除一条元组会“自动地”更新 S#_DURING、S_NAME_DURING、S_STATUS_DURING和S_CITY_DURING。

22.12 小结

在本章的开头提到了数据库中不断增长的包含历史数据的需求，只用时间戳来表示历史数据会导致一系列的问题——特别是很难处理某些约束和查询——而更好的方法是采用间隔表示。具体来说，引入了INTERVAL类型生成符和一些新的操作符来处理间隔数据。实际上，间隔和相应操作符不仅仅只使用于时态数据——尽管在例子中只涉及了INTERVAL(DATE)类型。在例子中涉及了时态关系（时态关系变量）和特定类型的属性。

间隔类型的定义以点类型为基础，同时要指定点类型的精度以及后继函数。

本章中讨论的操作符包括间隔操作符、间隔集合操作符和时态关系操作符。间隔操作符包括START、END和Allen操作符。间隔集合操作符包括UNFOLD和COALESCE（参见下一章）。时态关系操作符包括UNFOLD和COALESCE（同样参见下一章）。对于时态关系变量还有特定的更新操作符和约束（“时态码”）。使用这些新操作符和约束可以有效地处理时态数据。

同类型间隔集合上有两种重要范式，即展开形式和合并形式。如果集合中的每个间隔都是单位间隔，则INTERVAL（*PT*）类型的间隔集合是展开形式——即每个间隔中只包含一个点，每个点属于*PT*类型。如果任意两个间隔都不会覆盖或相连，则 INTERVAL（*PT*）类型的间隔集合是合并形式。这两种范式有助于避免某些冗余；合并形式是最简化的数据表达形式，

展开形式则是最便于操作的。将这两种范式扩展到包含间隔属性的关系上，则引入了重要的新关系操作符，UNFOLD和COALESCE，用来模拟关系投影和求差。

最后考虑一些数据库设计问题，对某些时态关系变量进行水平分解和垂直分解。

练习

22.1 a. 在SQL中，VARCHAR（3）表示不超过3个字符的字符串，缺省字符集为ASCII。你认为INTERVAL（VARCHAR（3））是可被接受的间隔类型吗？

b. 如果认为是的话，请用闭间隔表示['p','q']。

22.2 在第22.8节中定义了时态差操作符 I_MINUS。时态并（I_UNION）和时态交（I_INTERSECT）可以类似定义。请给出相应定义。

22.3 假设关系变量S_DURING上存在约束：在DURING上是合并的，而且假设需要更新使得从日期11到日期15供应商S1的状态为20。请给出相应的语句。不能采用第22.10节中扩展的更新操作符，但是可以认为S_DURING中包含了供应商S1在日期10的信息而且没有日期10以后的信息。S1在日期10的状态是任意的。

22.4 本章中讨论了如何将间隔操作符应用到时间间隔上，它们还可被应用到其它什么间隔？

22.5 给出具有多个（时态的或其它的）间隔属性的关系实例。

22.6 考虑关系变量S_DURING。在任意给定时间，如果存在供应商，则存在某个状态 *smax* 使得每个供应商的状态都不会超过 *smax*。使用本章讨论的操作符获取合并关系，其中每个等于*smax*的状态值和相应间隔。

22.7 在关系HW中有NAME、HEIGHT和WEIGHT属性，给出了每个人的身高和体重。书写一个查询来显示对于每个体重值满足以下条件的身高范围：在此范围中至少存在一个人的体重等于此体重值。

22.8 假设关系*R*中有两个间隔属性*I1*和*I2*。下面的命题是否正确，并证明：

a. (*R* UNFOLD *I1*) UNFOLD *I2* ≡ (*R* UNFOLD *I2*) UNFOLD *I1*

b. (*R* COALESCE *I1*) COALESCE *I2* ≡ (*R* COALESCE *I2*) COALESCE *I1*

22.9 能否给出一个具有间隔属性的关系例子，其中无须进行合并？

22.10 是否需要扩展“时态外码”，使之包含级联删除的功能？

参考文献和简介

22.1 J. F. Allen: “Maintaining Knowledge about Temporal Intervals,” *CACM 16*, No. 11 (November 1983).

22.2 Opher Etzion, Sushil Jajodia, and Suryanaryan Sripada (eds.): *Temporal Databases: Research and Pratice*. New York, N.Y.: Springer Verlag (1998).

1997年的论文集，同时也是进一步研究的最好参考资料。第四部分：常见参考文献中给出了综合目录和1998年二月的时态数据库概念常见术语表。第二部分：时态查询语言包括了“有效时间和事务时间建议：语言设计特征”一文，其中本章作者（Hugh Darwne）探讨了TSQL2中采用的方法，并指出在TSQL2标准中存在明显漏洞[22.4]。还包括了David Toman著的“基于点的SQL时态扩展及其有效实现”，建议对SQL基于点进行扩展，而非基于区间。这种思想引出了一些实现上的有趣问题。对这些问题的解答也

可能与基于区间的语言有关，因为 UNFOLD产生的单位区间“差不多”就是一些点（实际上，在IXSQL中它们的确是点——参见参考文献[22.3]）。

22.3 Nikos A. Lorentzos and Yannis G. Mitsopoulos: “SQL Extension for Interval Data,” *IEEE Transactions on Knowledge and Data Engineering 9*, No. 3 (May/June 1997).

本章中讨论的许多思想都是基于这篇论文。和 [22.2]一样，它也进一步引用了很多参考文献。

在提出对SQL扩展的建议之前，作者定义了扩展区间上的关系代数。它对 SQL的扩展称为IXSQL（有时读作“ nine SQL ”），不局限于时间区间。因为在 SQL中已经存在了关键字INTERVAL和COALESCE，作者提出了对应的 PERIOD和NORMALISE。如同[22.2]中提及的，IXSQL UNFOLD和我们提出的概念不同，它的结果是点的集合，而不是单位区间。Lorenttzos和Mitsopoulos因此还提出了一种逆 FOLD操作符，并在SELECT-FROM-WHERE子句中增加了 NORMALISE子句。 NORMALISE ON子句不仅必须写在 SQL语句尾部，同时也是最后执行的；也就是说， SELECT语句的输出是NORMALISE ON子句的输入。

22.4 Richard T. Snodgrass (ed.): *The Temporal Query Language TSQL2*. Dordrecht, Netherlands: Kluwer Academic Pub. (1995).

TSQL2是对SQL的时态扩展。特别地， TSQL2允许禁止对区间进行数量和关系操作。它不是简单地支持区间类型的生成和相关操作，而是提出多种特殊类型的表：快照表，有效时间状态表、有效时间事件表、事务时间表、双时态状态表和双时态事件表。

- 快照表是一般的SQL表，可能包括PERIOD时间类型的列。
- 其它表都支持时态：时态支持单个或两个时态元素的行级存在性。时态元素是一组时间戳，时间戳为PERIOD数值或日期时间类型的数值。

由PERIOD数值组成的时态元素应该被合并[⊖] 。时态元素不能作为常规列出现，但可以通过特定操作符访问。

下面是“具有时态支持”的各种表：

- 在有效时间表和事务时间表中，每个时间戳都是一个 PERIOD数值。
- 在有效时间事件表中，每个时间戳是某种日期时间类型的数值。
- 双时态表既是事务时间表，也是有效时间状态表或有效时间事件表。表中每一行有两个时态元素，一个表示事务时间，另一个表示有效时间。因此，双时态表可被视为事务时间表或有效时间表进行操作。

TSQL2中强调时态向上兼容性。这种思想允许在已经存在的基表上增加“时态支持”，从而将基表从快照表转换为某种时态表。这时，在基表上的各种常规 SQL操作都被解释为对其快照版本的操作，但会出现一些副作用。特别地，更新和删除当前的快照版本会同时保存这些行的旧版本。

TSQL2的最大优点是可以将那些序列化操作连接在一起。序列化操作是可以表示为对数据库快照上的操作，一般是在数据库的当前快照上操作，但实际上是会影响每个快照。例如，对有效时间表进行序列化操作的结果也是一个有效时间表。查询本身被表示

⊖ 1996年提交给ISO的TSQL2版本(未获批准)和[22.4]中提到的版本不同，其中具有时态支持的表总是“不可嵌套的”(即每个时态元素只能是单个时间戳，不能是一组时间戳)。也没有规定必须进行合并。

为只在当前数据库快照上查询，增加一个关键字来表明它是一个序列化查询。

不能表示为序列化操作的操作有时要用到比较秘密的语法。由于在 SQL中支持不含任何列的表，TSQL2强制规定具有时态支持的表中除了时态元素之外还必须拥有一个常规列。因此，对于查找在哪个周期中至少有一个巴黎的供应商已经签约这样的查询，无法表示为序列化形式。

部分练习答案

22.1 a. 可能不行（尽管可以这么认为）。 b. 'q'的前驱字符串要由可用的最大字符来决定，如果最大字符为'Z'，则答案为['p','pZZ']。注意：严格地说，VARCHAR(3)中的"(3)"不仅是数据类型的一部分，同时也是一种完整性约束。

22.2 首先，*R1* I_UNION *R2* ON *A*等价于

```
( R1 UNION R2 ) COALESCE A
```

在进行合并前无须对*R1*和*R2*在*A*上合并（为什么？）。然后，*R1* I_INTERSECT *R2* ON *A*等价于

```
( ( EXTEND ( R1 RENAME A AS A1 ) JOIN
           ( R2 RENAME A AS A2 )
           WHERE A1 OVERLAPS A2 )
   ADD ( A1 INTERSECT A2 ) AS A ) { ALL BUT A1, A2 } )
COALESCE A
```

在这里仍无须对*R1*和*R2*在*A*上合并。如果*R1*和*R2*在*A*上已经为合并形式，则最后的合并语句就可以去掉（为什么？）

其他关系操作符的时态版本（例如 I_JOIN）也可以类似定义。在第 22.10节中描述的UPDATE和DELETE可以作为时态版本使用。注意：在这里强调 I_MINUS的原因是它涉及展开操作，而其他时态关系操作符则不涉及，对于展开操作应尽可能避免。

22.3 下面是一个可能的解：

```
IF IS_EMPTY ( S_DURING WHERE S# = S# ('S1')
                       AND STATUS = 20
                       AND END ( DURING ) = d10 )
THEN INSERT INTO S_DURING
          ( EXTEND ( S_DURING WHERE S# = S# ('S1')
                              AND END ( DURING ) = d10 )
                                  { ALL BUT DURING }
            ADD INTERVAL ( [ d11, d15 ] ) AS DURING ;
ELSE UPDATE S_DURING WHERE S# = S# ('S1')
                     AND END ( DURING ) = d10
             DURING := INTERVAL ( [ START ( DURING ), d15 ] ) ;
```

22.4 可以按照接收光线和声音频率的范围，将动物划分为多个间隔。不同的自然现象可以按照发生的海拔位置划分间隔。下午 4点至5点喝茶是一个时态事实数据，但它显然与本章中讨论的内容相去甚远。毫无疑问你可以找到更多有意义的例子。

22.5 可以按照接收光线和声音频率的范围，将动物划分为多个间隔！另外，一旦我们将两个时态关系*R1*{*A*, *B*}和*R2*{*A*, *C*}连接起来，其中*B*和*C*为间隔属性，就得到了一个结果集合，虽然只是一个中间结果，但它拥有多个间隔属性。

22.6

```
WITH SP_DURING UNFOLD DURING AS SP_UNFOLDED :
( SUMMARIZE SP_UNFOLDED PER SP_UNFOLDED { DURING }
            ADD MAX ( STATUS ) AS SMAX ) COALESCE DURING
```

22.7
```
(( EXTEND HW { HEIGHT, WEIGHT }
   ADD INTERVAL ( [ HEIGHT, HEIGHT ] ) AS HR )
                                { WEIGHT, HR } ) COALESCE HR
```

22.8 命题a.显然成立，而命题b.则需要证明。令R如下所示：

I1	I2
[d01,d01]	[d08,d09]
[d01,d02]	[d08,d09]
[d03,d04]	[d08,d08]
[d04,d04]	[d08,d08]

(*R* UNFOLD *I1*) UNFOLD *I2*和(*R* UNFOLD *I2*) UNFOLD *I1*的结果为：

I1	I2
[d01,d01]	[d08,d08]
[d01,d01]	[d09,d09]
[d02,d02]	[d08,d08]
[d02,d02]	[d09,d09]
[d03,d03]	[d08,d08]
[d04,d04]	[d08,d08]

而(*R* COALESCE *I1*) COALESCE *I2*的结果为：

I1	I2
[d01,d02]	[d09,d09]
[d03,d04]	[d08,d08]

(*R* COALESCE *I2*) COALESCE *I1*的结果为：

I1	I2
[d01,d02]	[d08,d09]
[d03,d04]	[d08,d08]

下列命题也同样成立：

- *R* UNFOLD *I1*,*I2* ≡ (*R* UNFOLD *I1*) UNFOLD *I2*
- *R* COALESCE *I1*,*I2* ≡ (*R* COALESCE *I1*) COALESCE *I2*