

otorunr0Gr 00H o:0Tofo701erC1001 06 07010Y900 元

GnoY06奇3010100910101000江(0701010104Dr010020D 用 0 0 1 0

00 0100 你50 150 0 1 0aiemrcu 00:9:- 90103

**第** **2** **章** **OLTP 与数据服务**



数据服务 (Data Serving) 指的是面向各种操作型业务，提供数据的增加、删除、修 改以及简单的查询功能。提供数据服务功能的系统就是数据服务系统，包括支撑操作型业 务的关系数据库、NoSQL 数据库以及NewSQL 数据库等。

联机事务处理 (Online Transaction Processing,OLTP, 也称为在线事务处理),指 的是用户的业务请求转换成数据库的操作，传送到后台数据库管理系统，数据库管理系统 在很短的时间内，把用户的相关数据操作请求当作一个完整的事务来处理，对用户的请求 进行响应。联机事务处理有两个主要的特点： 一个是用户请求作为一个事务进行处理；另 一个是响应时间短。在这里，事务是一个完整的工作单元，要么全部执行，要么压根儿没 有执行 (All or Nothing)。事务的响应时间一般以秒计，比如我们在ATM 上进行账户余 额查询，从发起查询到查询结果返回，要求不超过3秒。

一般来讲，关系数据库管理系统(以及NewSQL 数据库)提供完备的联机事务处理 能力，保证数据库系统在执行用户事务后，数据是高度一致的、正确的。大量的操作型 NoSQL 数据库，其设计目标是具有极高的扩展能力，无法提供类似关系数据库管理系统 一样的事务处理能力，无法保障数据的强一致性 (NoSQL 数据库一般支持最终一致性， 不同副本的数据存在暂时的不一致)①,却可以提供关系数据库系统无法达到的系统规模 和吞吐能力。狭义上讲，关系数据库管理系统提供联机事务处理能力，操作型NoSQL 数 据库提供数据服务能力。广义上讲，可以把联机事务处理作为一类特殊的数据服务来看 待，数据服务中的数据操作也称为事务，只要一个数据操作或者一组数据操作从业务意义 上讲是一个完整的工作单元。我们按照广义来界定数据服务和联机事务处理的关系。

数据服务是相对于数据分析来讲的。两者的主要区别是数据服务一般在一个事务中只 需存取少量的记录，目的是反映操作型业务对数据的修改，比如新建订单、为订单付款、 账户余额的扣减、购物框的修改等。数据分析则往往需要在大量的数据之上(可以是整个 数据集),进行统计汇总以及更深层次的分析操作，目的是从大量数据中获得少量的汇总 指标和分析结果，以便从总体上对数据中隐藏的规律有所认识。



①事务、强一致性、最终一致性、事务特性和事务处理技术等内容，将在本章逐步展开。



**2.1面向OLTP应用的RDBMS数据库技术**

**2.1.1 关系数据库技术与SQL查询语言**

20世纪70年代初， IBM 工程师Codd 发表了著名的论文A Relational Model of Data for Large Shared Data Banks, 开启了数据管理技术的新纪元——关系数据库时代。关系 数据库管理系统 (Relational Database Management System,RDBMS) 就是在这篇论文的 基础上设计出来的。在此之前的数据库系统，主要有基于层次模型的层次数据库(比如 IBM 公司的IMS 系统)、基于网状模型的网状数据库(比如 IDS 数据库)等。这些数据库 的主要缺点是数据模型复杂，普通用户难以理解。另外，软件和数据模式 (Schema, 即 数据结构)的联系过于紧密。层次和网状数据库都采用导航式 (Navigational) 的数据结 构。存取特定的数据单元，必须在软件中按照一定的存取路径去提取数据，当数据模式发 生改变时，已经编写好的软件需要做相应的修改。

Codd 提出的关系数据模型，基于表格(关系)、行、列、属性等基本概念，这些概念易于 理解。比如，我们要保存一个学校的所有学生的基本信息，可以建立一张学生表，该表为一张 二维表。每个学生信息对应这张表的一行，也称为一个记录，或者一个元组 (Tuple) 。 每行数

**据由若干列组成，每一列表达了一个学生的一个属性，比如学号、姓名、性别、出生年月等** (**属性列也称为字段)。这样的二维表，不仅可以记录关于现实世界中的各类实体** **(Entity)** **的信** **息，而且可以记录实体间的关系** **(Relationship)。比如我们可以建立一张表格，这个表格具有学** **号、课程号、选课时间等属性列，表格的每一行记录了某个学生对某门课程的选课。**

在关系模型上，可以施加若干完整性约束条件，以保证数据的正确性、 一致性。完整性 约束包括实体完整性、参照完整性以及用户自定义完整性等。实体完整性是指数据库表的每 一条记录，都是和其他记录不同的唯一记录，比如一个学生和另外一个学生，他们的信息在 学生表里面是不同的两条记录。 一般通过唯一标识一行记录的主键 (Primary Key, 唯一标识 一行记录的若干属性，比如学生表的学号)来保证实体完整性。比如，在学生表中，虽然有 两个同学都叫王涛，但是他们具有不同的学号，通过学号我们可以在数据库里唯一地标识他 们。参照完整性是指如果某一个数据库表的记录，有一个字段参考了另外一个数据库表的另 外一条记录，那么另外那个数据库表的记录必须真实存在。比如学生表里包含一个属性列为他 所在的院系，该属性保存院系编号，参照了院系表，那么学生表的记录里出现的院系编号，必 须是院系表里存在的一条记录，即某一个具体院系的编号。用户自定义完整性，则是用户对数 据赋予的一些完整性规则，比如性别必须为男性或者女性，工资必须为一个正数等。

Codd 为关系模型建立了严格的关系代数运算，这些运算也称为关系操作。主要的关系 操作包括选择、投影、连接等。选择就是把一张表格中符合条件的记录挑选出来，比如把 1997年9月以后出生的学生记录选择出来。投影是在表格上把各个记录的部分属性列提取出 来，比如我们在学生表格上进行投影操作，只显示学生的学号、姓名和性别信息。连接操作 则把两张数据库表或者多张数据库表，按照一定条件，把它们的各一行记录连接起来，生成 结果集的一条记录。比如，我们对学生表、课程表、选课表进行连接查询，显示学生的学 号、姓名、课程号、课程名等属性列，表示不同学生对不同课程的选课信息。又比如，我们



对学生表和院系表进行连接查询，显示每个学生的学号、姓名以及所在院系的名称。

下面我们针对学生表和院系表上的连接查询，解释为什么需要进行表格之间的连接操 作，以及如何实现连接操作。查询的SQL 语句如下：

|  |
| --- |
| **Select** s.sno,s.sname,d.dname  **From** student s,department d **Where** s.did=d.did |

注 ：student 为学生表的名称，s 为其别名；department 为院系表的名称，d 为其别名；s.did=d.did 为两个表的等值连 接条件。

该查询需要显示学生的学号、姓名，这些信息在学生表里面。此外，该查询还需要显示 学生所在院系的名称，这些信息在院系表里面，因此，需要对学生表和院系表进行连接。可 能有的读者会问，为什么我们不能把学生所在院系名称保存在学生表的某个字段里面，进行 上述查询时，就不需要两个表格之间的连接了?这是可以的，但是这会带来严重的问题，那 就是如果我们在学生表的每个学生的院系字段里保存了院系名称，院系名称比起院系ID 来 讲，占用的空间大得多，当学生数量很多时，浪费的空间是很大的。此外，当院系改名时， 我们需要对隶属该院系的所有学生的院系字段进行修改，代价很大。对关系数据库的表格进 行设计时，特别是面向OLTP 类应用的数据库中，我们应该尽量把不同的实体放在不同的表 格中。不同表格中的实体，通过主外键关系建立联系，这个设计过程称为规范化 (Normali- zation), 具体细节请参考相关资料。比如，学生表的每个学生记录通过学号唯一标识，学号 是学生表的主键。院系表的每个院系通过院系ID唯一标识，院系ID 是院系表的主键。学生 表的院系id 字段是一个外键，建立了学生表和院系表之间的联系(见图2-1)。

|  |  |
| --- | --- |
| **院系表** | |
| 院系ID | 院系名称 |
| 001 | 信息学院 |
| 002 | 经济学院 |

|  |  |  |  |
| --- | --- | --- | --- |
| 学生表 | | | |
| 学号 | 姓名 | … | 院系ID |
| S001 | 张三丰 |  | 001 |
| SO02 | 赵敏 |  | 001 |
| S003 | 张无忌 |  | 002 |
| SO03 | 周芷若 |  | 002 |

**图2-** **1** **学生表和院系表的连接**

实现连接的算法有三个，分别是嵌套循环连接 (Nest Loop Join)、排序合并连接 (Sort Merge Join) 和哈希连接 (Hash Join)。嵌套循环连接，以学生表和院系表的连接为 例，我们从院系表里提取一个院系记录，然后扫描整张学生表，把隶属该院系的学生记录 和该院系记录连接起来输出，再提取下一个院系，接着针对该院系，扫描整张学生表…… 如此往复，直到院系表的所有记录处理完毕，连接操作完成。排序合并连接，我们针对院 系表，基于院系ID 字段(就是SQL 查询语句的连接字段)进行排序，然后针对学生表， 基于院系 ID 字段进行排序。这时候，院系表和学生表在院系ID 字段上的排序是一致的， 只不过一个院系对应若干学生(如图2-1所示)。接着我们用两个光标 (Cursor) 顺序扫 描两个表，当遇到院系表的一个记录，就在学生表里面把可以和它连接的记录都与之连 接。然后院系表的光标移动到下一个院系记录，学生表的光标移动到隶属于该院系的第一 个学生记录……直到两个表的光标都达到表格的末尾，连接操作完成。哈希连接是使用一 个 Hash 函数 (Hash 函数的功能是把数据映射到一个地址，在这里把院系ID 映射为一个 桶 (Bucket) 号),扫描院系表的院系记录和学生表的学生记录，针对院系ID 进行Hash 操

作，把院系记录和学生记录分配到不同的桶(桶是一种内存数据结构)。由于使用同一个 Hash 函数，于是某个院系记录及其所属学生记录被分配到同一个桶。比如信息学院和信 息学院的学生记录被分配到同一个桶，经济学院和经济学院的学生记录被分配到另外一个 桶。我们就可以在一个桶之内，对匹配的院系记录和学生记录进行连接。

关系模型提出来后，研究人员掀起了关系数据库管理系统开发热潮。特别值得提及的 两个原型系统，一个是 IBM 开发的System R,另一个是加州大学伯克利分校开发的 In- gres 。1974 年作为System R 项目的一部分，IBM 的工程师开发了交互式查询语言SE- QUEL(Structured English Query Language), 这是SQL 语言的前身。

现在，SQL 语言已经成为国际标准，成为各个数据库厂家产品的标准的数据定义语言、 查询语言和控制语言。通过标准化的SQL 语言，用户可以进行模式(即表格的结构)和索 引的定义与删除，对数据库表的数据进行增加、删除、修改以及查询等操作。使用SQL 语 言可以表达复杂的查询操作，我们可以对一张数据库表进行查询.也可以对多个表进行连接 查询，还可以进行嵌套查询。比如我们可以把符合某个条件的院系查找出来.然后根据查询 结果，把隶属于这些院系的所有学生查找出来。在查询之上，还可以对数据进行分组、聚集 操作，比如按照院系统计学生的人数，把各个职称序列的老师的平均工资求出来等。

虽然功能如此强大，但 SQL 语言非常容易理解，普通用户经过简单培训就可以掌握 和使用。SQL 语言是一种声明性的语言 (Declarative Language)。使 用SQL 语言，用户 只需告诉系统查询目的是什么(需要查询什么数据),即What, 并不需要告诉系统怎么样 去做，即 How, 包括数据在磁盘上是怎么存储的，可以使用什么索引结构来加快数据访 问，以及使用什么算法对数据进行处理等，都无须用户关心。比如，从学生表、课程表、 选课表中把某个学生的选课信息提取出来的SQL 查询，其形式如下：

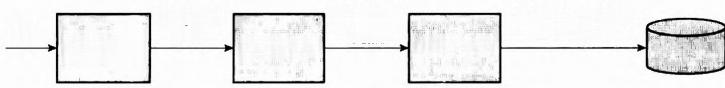
|  |
| --- |
| **Select** s.sno,s.sname,c.cno,c.cname  **From** student s,course c,student\_cource sc  **Where** s.sname=" 张三"and s.sno =sc.sno and  c.cno =sc.cno |

注：student 为学生表的名称，s 为其别名；course 为课程表的名称，c 为其别名；student\_cource 为选课表的名称，sc 为

其别名；s.sname= “张三”为学生表的过滤条件；s.sno=sc.sno and c.cno=sc.cno 为三个表的等值连接条件。

一般来讲，一个SQL 查询语句，要经过词法分析、语法分析、语义检查，在内存中 生成一棵语法树，经过优化器优化后生成一棵优化的语法树，再转换成物理执行计划，最 后由执行器执行，获得结果集，如图2-2所示。查询执行计划 (Query Execution Plan) 是由操作符构成的一棵语法树。操作符指的是选择、投影、连接、聚集等操作。查询执行 计划分逻辑执行计划和物理执行计划。逻辑执行计划指的是SQL 语句经过词法分析、语 法分析和语义检查之后，生成的语法树，以及经过优化器在逻辑层面优化(请参考下文基 于规则的优化器)之后生成的语法树。物理执行计划则是优化器根据数据的特点，确定了 每个操作符的具体实现的语法树，可以直接交给查询执行引擎 (Query Execution Engine) 运行，获得查询结果。确定了具体实现的操作符，比如对于表格数据的提取，是对整张表 进行扫描，还是通过索引进行部分数据的提取，称为物理操作符 (Physical Operator)。

上述Student,Course,Student\_Course 三表连接的SQL 查询，经过词法分析、语 法分析和语义检查之后，生成的执行计划如图2-3所示。按照图2-3上各个圆角矩形内

Logical Physical

Storage engine interface

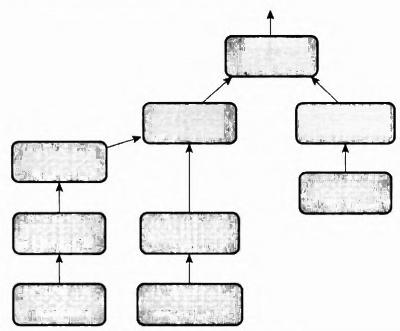
Plan Optimizer Plan **Execution**

SQL

SQL Parser

**图2-2** **SQL** **语句的执行过程**

的编号，我们对语法树进行解释如下：(1)表扫描操作符 (Scan) 扫描S 表；(2)过滤操 作符，根据s.sname= “张三”的查询条件，对S 表的记录进行过滤；(3)由于最后我们 只需显示学生的学号和姓名，投影操作符对S 表的记录进行投影，只提取学生的学号和姓 名；(4)表扫描操作符 (Scan) 扫描 SC 表；(5)为了进行连接操作，我们只需SC 表 的 sno 和 cno 字段，投影操作符对SC 表进行投影，只提取这两个字段；(6)对操作符3和操 作符5递送来的S 表记录和 SC 表记录，进行连接操作，生成临时结果；(7)表扫描操作 符 (Scan) 扫 描C 表；(8)对于C 表，由于最后我们只需显示课程名，但是和SC 表进行 连接需要课程号，投影操作符对C 表的记录进行投影，只提取其课程号 cno 和课程名 cname;(9) 对操作符6和操作符8分别递送来的S 表 和 SC 表连接的中间结果以及C 表 数据，进行连接操作。最后把结果返回给用户。



9.Join

6.Join 8.Project

3.Project

7.Scan

2.Filter 5.Project

1.Scan S 4.Scan SC

**图2-** **3** **查询执行计划**

关系数据库管理系统的查询优化器，根据用户的查询特点和数据的分布特征，选择合 适的查询执行计划 (Execution Plan),通过过滤、投影、连接、聚集等操作，完成用户的 查询，力图达到执行速度快、消耗资源少、尽快获得查询结果等目标。查询优化器是关系 数据库管理系统最重要的和最复杂的模块之一，经历了从简单到复杂、基于规则到基于代 价模型的发展阶段。

基于规则的优化，就是根据一定的规则，优化查询语法树。比如对于上述查询来讲， 查询的执行过程可以是先对学生表、课程表、选课表进行连接，然后选择符合条件的学 生，最后进行投影操作，把需要的属性列提取出来。基于规则的优化，往往把选择操作和 投影操作放在连接操作之前完成，以便进行连接操作时，参与连接操作的数据量已经大大 减少，从而提高查询效率。比如在上述实例中，我们可以把张三这个学生从学生表中过滤 出来，然后和课程表以及选课表进行连接。

基于代价的查询优化，则需要优化器掌握数据的统计特征，适当选择合适的查询处理 算法。比如在多表连接操作中， 一般是两个表先进行连接操作，生成中间结果集，然后中 间结果集和后续的其他表进行连接操作， ……,依此类推。在这种情况下，不同的表间连

接顺序 (Join Order) 对性能的影响很大。比如上述实例中，学生表S 、课程表 C 和选课 表 SC 三者的连接顺序有6种方式，即三个表的全排列，即S/C/SC (该连接顺序无效), S/SC/C,C/S/SC (该连接顺序无效),C/SC/S,SC/S/C,SC/C/S 的连接顺序。一般来 讲，优化器可以根据数据的统计特征，估计不同的连接顺序的中间结果集的大小，选择合 适的连接顺序，尽早减少中间结果集的大小，从而加快后续的连接操作。当有多个索引可 以使用时，到底使用哪个索引加快数据的访问；进行数据库表之间连接，可以选用不同的 连接算法，到底使用哪个连接算法，都由优化器来确定。

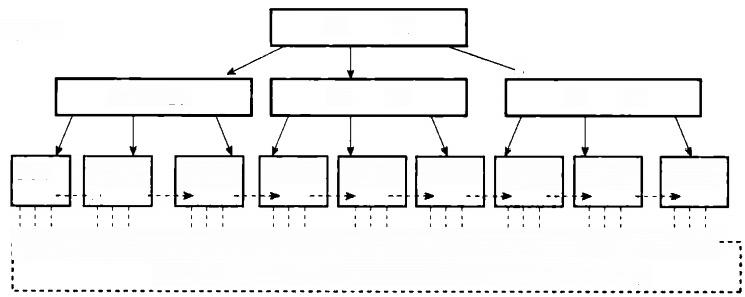
20世纪70年代以来，围绕关系数据库技术的公司和产品部门纷纷创立。大浪淘沙， 其中获得商业成功的公司(部门)主要有 IBM(DB2) 、(Oracle(Oracle) 、Informix 、Sy-

base、微软 (SQL server)、SAP等。这些数据库技术公司创造了庞大的数据库产业，每 年创造巨大的产值。关系数据库技术是当前主流的数据库技术，支持大量的联机事务处理 应用。关系数据库产品，负责保存和管理银行、航空等和人们日常生活息息相关的关键业 务系统 (Mission Critical) 的数据，夜以继日执行各种事务，支持业务的运转及社会的 运 行 。

**2.1.2** **利用索引加快数据访问**

当我们要查找字典里的某个字时，一般需要通过偏旁部首或者拼音来查找，而不是从 头到尾在字典里查找这个字。拼音表和偏旁部首表就是字典的索引。在数据库里.索引是 一种辅助数据结构.它可以帮助数据库查询引擎加快对数据的访问。在数据库技术蓬勃发 展的几十年中，人们提出了各种各样的索引技术。这里，我们通过实例，简要介绍其中应 用非常广泛的一种索引技术——B+ 树索引。

图2-4展示了一个3阶的B+ 树(3阶指的是B+ 树的非叶子节点有三个指针)。假设 有一个用户表，包含若干用户记录，每个记录包括用户ID、 姓名、性别、出生年月、电 话、邮编、地址等信息。目前，表格中的用户 ID 有 … , 7 , … , 1 6 , … . 2 5 , … , 34, … ,43, … ,64, … ,75, … ,88, … ,97等，我们在用户ID 字段上建立 B+ 树索 引，加快基于用户ID 对数据进行查找。建立的索引如图2-4所示。



B+树索引的内部结构 30 70

▲

10 20 40 50 80 90

5/67 1415/16 23724/25 32/3334 414243 62/63/64 737475 868788 959697

5167---i4151672372425…32334414243-7626364737475-7868788795969 注 意 ：B+树叶子节点的指针指向这些键值对应的数据库记录

B+ 树索引

的叶子节点

数据文件

**图** **2** **-** **4** **一** **个** **3** **阶** **的** **B+ 树**

B+ 树索引组织成一棵树的结构，它包含非叶子节点和叶子节点，每个节点包含一系 列的Key 和一系列的指针。非叶子节点的指针指向其他非叶子节点或者叶子节点，叶子节 点的指针指向数据文件(数据库数据存在文件里)的具体记录。 B+ 树有一个对数据进



行访问的人口，即根节点。

我们通过图2-4来具体认识一个B+ 树的特点。在该B+ 树中，根节点包含30,70 两个 Key (键值),三个指针分别指向三个非叶子节点。左边的指针指向的节点，所有的 Key 值小于70,中间的指针指向的节点，所有的 Key 值落在30～70之间，右边的指针指 向的节点，所有的 Key 值都大于70。我们再看左边那个非叶子节点，它包含2个Key 值 ， 分别是10,20,以及三个指针，左边的指针指向的节点(叶子),所有的 Key 值都小于 10,中间的指针指向的节点(叶子),所有的Key 值都落在10～20之间，右边的指针指向 的节点(叶子),所有的 Key 值都大于20,到这里我们就可以观察到 B+ 树各个节点的 Key 值和指针的组织形式以及它们的关系。

当我们寻找某个具体的Key 值时，沿着根节点，经过跟每个节点的一系列 Key 值 的 比较，沿着某个指针前进，经过一系列的非叶子节点，到达叶子节点，最后可以通过叶子 节点的指针，定位到数据文件的某个记录。在大量的数据中寻找某个记录的操作，速度大 大提高了。

B+ 树的组织形式总结如下：(1)叶子节点的组织形式： B+ 树的查找键是数据文件 的主键，而且索引是稠密的，即每个记录在索引字段上的值，都在索引中出现。叶子节点 中为数据文件的每个记录设有一个<键值、指针>对，指向数据文件的某条记录。(2)非 叶子节点的组织形式： B+ 树的非叶子节点，形成叶子节点上的一个多级稀疏索引。每个

非叶子节点中至少有ceil )(表示大于或者等于 的最小的整数)个指针，最多有m 个 指针。根节点是一个特殊的节点，所有的数据查找操作，都从根节点开始。

在实际应用中，每个非叶子节点的大小以磁盘块为大小，可以容纳上千个键值和指 针。也就是B+ 树的阶数达到上千的级别，即便对于记录数量较大(上亿)的表格， B+ 树的深度也只需2～5个层级。每次磁盘操作，可以存取一个非叶子节点，少数几个磁盘 I/O 就可以定位到具体的数据。通过观察B+ 树的结构，我们还看到，兄弟叶子节点之间 是连接起来的。当进行范围查询时，比如范围是 [Key,Key₂], 当定位到包含Key 的叶

子节点以后，就可以沿着这个链条寻访符合范围查询条件的其他记录键值所在的叶子节 点。通过叶子节点的指针，可以访问数据文件的具体记录。由此可以看出， B+ 树不仅可 以加快点查询 (Point Query, 即查找某个 Key), 而且可以加快范围查询 (Range Query)。

2.1.3 数据库的事务处理、恢复技术与安全保证

大量的业务数据通过数据库来进行管理，数据的一致性和可靠性直接影响业务的开 展。为了保证数据库数据的一致性(数据库数据处于有效的状态或正确状态)和数据库系 统的可靠性，人们提出了事务的概念，以及并发控制和恢复技术。

数据库里的事务指的是一组数据库操作，这些操作是互相关联的操作，目的是完成一 定的业务目标。比如转账事务，目的是把一个账户的一定金额，转入另外一个账户，它包 含两个数据操作，分别是减少一个账户的余额，以及增加另外一个账户的余额。为了保证 数据的 一致性，事务处理器 (Transaction Manager) 必须保证事务的四个主要特性 ACID, 即事务的原子性 (Atomicity) 、 一 致 性 (Consistency) 、 隔离性 (Isolation) 和持 久性 (Durability)。

事务的原子性指的是事务的所有操作，要么全部执行，要么都没有执行 (All or Nothing)。 比如执行一个转账事务，要么已经完成转账，要么未完成转账，不允许出现已

经从一个账户扣钱，钱没有到达另外一个账户的状况。事务的一致性指的是事务把数据库 状态，从一个有效状态转化为另一个新的有效状态，在事务失败的情况下，必须把所有数 据恢复到事务开始之前的状态。事务的隔离性指的是在一个并发的事务处理系统中，多个 事务的各个操作步骤可以交替执行，但是必须保证某个未提交的事务和其他事务是相互隔 离的，目的是保证未提交的数据，别的事务不能看到。事务的持久性指的是提交的数据必 须保存起来，当系统失败或者重启，数据能够恢复到最近的正确状态。

事务的ACID 特性，以及整个数据库系统的可靠性，依靠并发控制技术和恢复技术来 实现。数据库的并发控制技术主要分为两类，包括基于加锁的并发控制技术，以及多版本 并发控制技术 (Multi Versioning Concurrency Control)。以最简单的加锁技术为例，当事 务对数据进行读取时，需要对数据加读锁，当事务将要写入数据时需要对数据加写锁。在 同一数据上，不同事务的读锁可以并存，但是同一时间只能有一个事务拥有写锁，这样就 能保证事务的正确协调。基于多版本的并发控制技术，读事务读取数据的最新提交版本 (Committed Version),针对写事务的每个写操作，数据库系统都从原有数据拷贝一个新 版本，并且一直由该事务维护数据的这个未提交版本 (Un-Committed Version), 等到事 务提交时，才把它固化为最新的提交版本。如果多个事务并发修改同一数据，那么在它们 提交时.需要进行冲突解决，某些事务需要退出 (Abort) 。 如果事务退出，未提交的数据 版本将被丢弃掉。

数据库恢复技术，保证数据库系统在失败后能够恢复到最近的一致状态。为了达到这 个目标，数据库的所有操作都必须有所记载，这就是数据库日志。日志记录的基本原则是 “先写日志原则”(Write ahead Logging), 即对数据进行操作之前.把必要的日志信息记 录下来，这些信息一般记录数据改变之前是什么值(前像， Before Image), 改变之后将是 什么值(后像， After Image) 等。数据库的操作一般很频繁，日志数据量急剧增加，占用 大量磁盘空间，也导致数据库恢复时需要扫描处理的日志信息过多。 一般通过检查点技 术.把数据库的最近完整状态整体保存起来，日志文件里仅仅需要记录检查点以来的事务 对数据的改变就可以了，大大减少日志数据量，能够加快恢复过程。在数据库恢复过程 中，首先把检查点装载进来，把提交的事务再执行一遍 (Redo). 把未提交事务对数据的 改变恢复回去 (Undo), 就可以把数据库恢复到最近的一致状态。为了应付存储介质损 坏，把数据库数据保存到可靠的存储器，称为数据库的一个备份。对数据库的所有数据进 行完整备份，称为全量备份。对数据库上次备份以来修改过的数据进行备份，称为增量 备 份 。

数据库里的数据，如果没有施加任何保护，就可能被窃取或者篡改，数据库的安全性 是一个重要的问题。数据库的安全性保证，主要的技术手段包括用户认证和授权、审计、 数据加密等。用户认证是对存取数据库的用户，鉴定其身份。用户授权则根据用户的角 色，授予其存取数据的权限。现代数据库可以对不同用户对不同级别的数据库对象的不同 操作，分别进行授权。比如可以对整张表的操作或者字段级的操作进行授权。审计是对用 户在数据库数据上的操作进行记录，以便出现问题时可以追本溯源。数据加密是把数据进 行加密后保存到数据库表中。加密的数据需要解密才能查看。由于加密和解密需要付出计 算开销，一般来讲，只有敏感的数据才有必要进行加密，比如职工的工资字段为敏感数 据，需要加密存储。除此之外.数据库的视图机制提供了一定的安全保证。在数据库表之 上，通过查询(选择、投影、连接等)定义一个视图，然后把视图的存取权限授予部分用 户，一定程度上增强了数据库的安全性。因为视图向外展现的数据只是整个数据库表的一 个行列子集，用户不能存取视图之外的数据库表里的其他数据。



**2.1.4** **并行数据库与分布式数据库**

为了提高数据库性能，人们研发了并行数据库技术。并行数据库指的是通过高速网络 把多个计算节点连接起来，把数据库数据和操作分解到这些节点上并行执行的一类数据库 系统。有面向OLTP 应用的并行数据库.也有面向(LAP 应用和数据分析应用的并行数 据库，以及支持(LTP/OLAP 混合负载的并行数据库。关于并行数据库的架构、数据分 片、查询处理等内容，将在第3章“OLAP 与结构化数据分析”统一介绍。

分布式数据库指的是这样的数据库系统，它运行在多台计算机上，这些计算机通过网 络互联。每台计算机可以放在一个地方，每台计算机安装独立的RDBMS, 拥有数据的完 整拷贝或者部分拷贝。这些计算机系统共同组成一个完整的、逻辑上集中，但是物理上分 散的大型数据库。归结起来、分布式数据库的主要特点是数据物理分布在各个场地，但它 们逻辑上是一个整体。各个场地上的数据由本地RDBMS 管理，该RDBMS 具有自治处理 能力。各个场地虽然具有高度的自治性，但是需要相互协作，构成整体，完成全局事务 处 理 。

并行数据库和分布式数据库的主要区别是并行数据库的每个节点并未安装独立的 RDBMS, 只是整个RDBMS 的一部分，其管理的数据也是整个数据库的一个子集，节点 间的连接一般通过高速网络.以便支持高速的数据访问。

相对于传统集中式的数据库来讲，分布式数据库具有如下优点：(1)分布式数据库把 各场地自治管理和集中式数据查询及操作的要求巧妙结合起来。(2)它具有更高的数据访 问速度，分布式数据库为了保证数据的高可用性，一般采用多副本存储的容错策略。在进 行数据读取时.不同客户(端)可以从不同副本读取所需要的数据，提高数据的访问速 度。(3)更高的并发能力，分布式数据库采用计算机集群实现，可以提供更高的用户并发 访问量。(4)更高的系统扩展能力.分布式数据库可以通过增加节点来实现系统的扩展， 相对于集中式数据库具有更高的扩展能力。

**2.2** **面向数据服务的NoSQL数据库技术**

**2.2.1 NoSQL 数据库技术①**

NoSQL 数据库并不是某个具体的数据库系统，而是一类数据库系统的统称。近年来， 各类NoSQL 技术异军突起，蓬勃发展。其主要特点是采用与关系模型不同的数据模型， 在系统设计时，面向大数据处理的新挑战，采取了一些新的设计原则，目的是利用大型的 计算机集群实现大数据的有效处理。

这些新原则包括：(1)采用横向扩展 (Scaling Out) 的方式，应对大数据处理的挑 战，通过大量节点的并行处理，获得极高的数据处理性能和吞吐能力，包括读性能和写性 能 。NoSQL 数据库需要对数据进行划分 (Partitioning), 以便进行并行处理。(2)放弃严

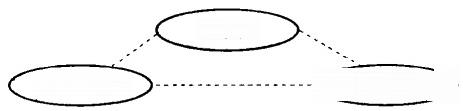
① 传统的关系数据库管理系统也称为SQL 数据库，因为它们都基于关系数据模型，提供了SQL 接口。

格的ACID 一致性约束，采用放松的一致性约束条件，允许数据暂时出现不一致的情况， 接受最终一致性 (Eventual Consistency) 。(3) 对数据的存储进行容错处理， 一般对数据 块进行适当备份(比如在不同节点上维护数据的三个副本),以应对节点失败状况，保证 在由普通服务器组成的集群上稳定可持续地运行。

下面我们将按照四个类别介绍NoSQL 数据库技术。对于每个类别，介绍一两个典型 的 NoSQL 数据库系统，剖析其技术特色。

**2.2.2 CAP 理** **论**

根据Brewer 提出的CAP 理论(后来由Gilbert 和 Lynch 证明),在大型分布式系统 中，一致性 (Consistency) 、 系统可用性 (Availability) 和网络分区容忍性 (Network Partition Tolerance) 这三个目标中，只可以获得其中两个特性。追求两个目标将损害另 外一个目标，三个目标不可兼得(如图2-5所示)。换言之，如果追求高度的一致性和系 统可用性，网络分区容忍性则不能满足。关系数据库一般通过保证事务的ACID 特性实现 数据的一致性，并且通过分布式的事务执行协议，比如两阶段提交协议① (Two Phase Commit,2PC) 等保证事务的正确执行，追求系统的可用性，于是丧失了网络分区容忍 性。在大量节点组成的集群系统中，由于节点失败是一个普遍现象，有可能造成数据库查 询不能正确完成，不断重启.永远无法结束的情况。



一致性

系统可用性 网络分区容忍性

**图2** **-** **5** **CAP理论**

ACID 实施了强一致性 (Strong Consistency) 约束，使得关系数据库系统很难部署到 大规模的集群上(几千个节点规模)。 NoSQL 数据库则通过放松一致性的约束，把数据和 处理任务分布到大量的节点上运行，它追求系统可用性和网络分区容忍性，但是牺牲了一 致性。

**2.2.3 Key-Value 数据库**

**1.Dynamo 数据库**

Dynamo 数据库是Amazon 开发的键值对数据库。② Key-Value 数据库是数据库里的每 个记录包含两个部分，分别是主键 Key 和值 Value, 在 Value 部分可以存放任意数据。 Key-Value 数据库的数据模型很简单，它可以方便地支持各类上层应用。

Dynamo 采用一系列技术，实现了高性能的、可扩展的和高可用的 Key-Value 数据 库。其99.9%的读写访问，可以在300ms 内完成。Dynamo 是第一个具有极大影响力的 NoSQL 数据库系统，成为其他NoSQL 数据库模仿的对象。

Dynamo 使用一致性哈希(Consistent Hash) 技术，实现数据的划分和分布。这个技



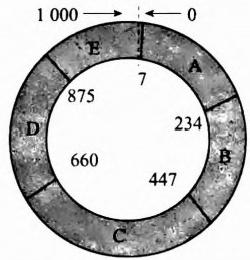
①两阶段提交协议将在第3章介绍。

②在正文中，键值对数据库也称为Key-Value数据库，键值对本身则表示成<Key.Value>.



术的基本原理是，使用一个 Hash 函 数H, 把 Key 值均匀地映射到一系列整数中， H (Key)=Key mod L运算 (mod 表示取余运算)就能把 Key 值映射到[0,L] 上。把0 和L 首尾相连，形成一个环。在图2-6中，服务器A 负责所有 Hash 值落在[7,233] 的 Key 值的管理， ……,服务器E 则负责 Hash 值落在[875,6]的Key 值的管理。

为了对数据进行复制以支持更高的容错性，可以把数据保存到负责邻近Key 范围的节 点上。比如，当复制因子 (Replication Factor) 为3时，所有映射到[7,233]的 Key 和 对应的数据，被保存到A,B,C 这3个节点上。当某个节点负载过重时，可以往集群里 增加节点(即往虚拟环中增加节点),一致性Hash 的映射关系能够保证只需迁移少量的 数据，即把在环上和新增节点相邻的节点的部分数据进行迁移即可。



**图2-6** **一致性Hash 以及键值[0,1000]和节点** **[A,E] 的对应关系**

Dynamo 使用Quorum 机制(也称为NRW 方法),实现数据的容错备份，保证数据的 一致性和系统的可用性。N 为副本(也称备份)的个数， R 为读数据的最小节点数，W 为 写成功的最小节点数。通过这三个参数的配合，可以灵活调整Dynamo 系统的可用性与数 据一致性。

我们通过实例来看它的运行机制。比如N=3 &.R=1&.W=1,表示最少只需从一个 节点读取数据即可，读到数据就可以返回，进行写入时，只要在N 个副本里写入其中一个 即可返回。这时候系统可用性很高，但是并不能保证数据的一致性，也就是读取的数据可 能不是刚刚写人的数据。当N=3&.R=3&.W=3 时，每次写操作，都需要保证所有的 副本都写成功，读的时候，也需要从所有的副本读取数据才算读成功。这样读取出来的数 据一定保证正确性，但是由于读写过程中需要涉及3个副本，性能大受影响。数据的一致 性得到保证，但是系统可用性和性能降低了。采用N=38.R=2 &.W=2 的读写模式， 是对上述两种极端情况的折中，既保证了数据的一致性，又保证一定的系统可用性和性 能。这种模式的本质是 R+W>N, 它能够保证读取得到的数据肯定是刚刚已经写入的数 据，读取的份数一定要比总的副本数减去确保写成功的副本数的差值还要大。也就是说， 每次读取，都至少读取到一个最新的版本，从而保证我们能够“读我们所写”。

除此之外，Amazon 还使用如下技术，保证了Dynamo 数据库系统的鲁棒性。Dynamo 通过 Hinted Handoff机制，在一个节点出现临时性故障时，把写操作自动引导到节点列 表的下一个节点进行，并标记为Handoff 数据，在收到通知需要对原节点进行恢复的消息 时，重新把数据推回去。这个机制使得系统的写人成功率大大提升。

Dynamo 使用向量时钟 (Vector Clock) 技术，实现版本冲突处理。它用一个向量表 示数据的不同版本，在版本冲突的情况下，可以追溯出现问题的地方，保证了系统的最终 一致性。每个节点都记录自己管理的数据的版本信息，也就是每个数据都包含所有的版本

**信息。读取操作返回多个版本，由客户端的业务层来解决这个冲突，合并各个版本。**

我们通过一个实例了解Dynamo 是如何把Quorum 机制和向量时钟技术结合起来，实 现版本冲突处理的。假设整个集群有 A,B,C 三个节点，系统使用的复制因子 (Replica Factor)为3,即每个数据有3个副本。

当采用W=1 时，为了保证W+R>N, 有 R=3, 那么有如下场景：(1)A 收到某个数 据X 的数值是4000的写请求，于是对于该数据，其数据和版本信息为4000 [A/1]。(2) 数据被复制到B,C 前，该数值被调整，变成4500,那么A 上有4500 [A/2], 覆盖了 4000 [A/1] 。(3) 这个数据被复制到B,C 两个节点，B,C 上有4500 [A/2] 。(4) 这时 候，对节点B 有个更新请求，X 被改成5000,那么B 上就有5000 [A/2.B/1] 。(5) 在

B的数据被复制到A,C 之前，对节点C 有一个更新请求，重新改成3000,于是C 上有 3000 [A/2,C/1]。于 是 A,B,C 三个节点的数据 X 的版本标记为4500 [A/2],

5000 [A/2,B/1],3000 [A/2,C/1]。

当客户端读取时，三个节点读来的数据不一致。由于我们的R 设置为3,所以读到了 这三个版本的数据，A 的版本最低，被舍弃，B 和 C 的数据5000 [A/2,B/1],3000

[A/2,C/1] 需要进一步甄别哪个是最新的。可以通过时间戳来比较，比如 B 上的数据是 最新的，则X 的取值是5000。确定了最新数据后，合并向量时钟，通知节点A 和C 把数 据改成5000 [A/3,B/1,C/1]。 后续的读取操作，都从三个节点读取，比较三个数据版 本，这时A 上的X 数据版本最高，为最新数据。

当采用W=2 &.R=2读写配置时，有如下的场景：(1)A 收到对X 的写请求4000, 这个数据必须到达 B 才算写成功。于是 A 上有4000 [A/1],B 上有4000 [A/1]。

(2)在数据X 被复制到C 之前，有个对X 的改变，变成4500,同上， A 上有4500 [A/ 2],B 上有4500 [A/2]。(3) 数据被复制到C,C 上有4500 [A/2]。(4) 这时对 B 有一 个对X 的修改请求，变成5000,那么B 上有5000 [A/2,B/1], 复制第二份到C, 于 是 C 上有5000 [A/2,B/1] 。(5) 对C 有一个对数据X的修改请求，变成3000,那么C 有 3000 [A/2,B/1,C/1], 数据复制第二份到A,A 上的4500 [A/2] 相对于3000 [A/

2,B/1,C/1] 更陈旧，被更新的数据覆盖，变成3000 [A/2,B/1,C/1]。

这 时 ，A 上有3000 [A/2,B/1,C/1],B 上有5000 [A/2,B/1],C 上有3000

[A/2,B/1,C/1] 。 由于R=2, 无论我们读哪几个数据，最后都得到5000 [A/2,B/1]

和3000 [A/2,B/1,C/1]。 版本 [A/2,B/1,C/1] 要比版本 [A/2,B/1] 更新，于

是在W=2&.R=2&N=3 的情况下，无须协调即可解决版本冲突。

需要指出的是，通过提高 W 可以降低冲突，提高一致性，但是写入多份数据要比写 入一份数据慢，写成功的概率也降低了，降低了系统的可用性，这也印证了CAP 理论。

最后，Dynamo使用Gossip 协议，实现成员资格和错误检测。使用该协议.整个网络 省略了中心节点，使得网络可以去中心化，提高了系统的可用性。

**2.Redis 数据库**

Redis 数据库是一个开源的、可以基于内存也可以持久化的、高性能的Key-Value 数 据库。Redis 使用ANSI C语言编写，提供了各种语言的应用程序编程接口 (Application Programming Interface,API)。

与其他Key-Value 数据库不同的是，Redis 的值 (Value) 的类型不仅仅限于字符串， 还支持其他数据类型，包括字符串列表 (String) 、 哈希表 (Hash) 、 列表 (List) 、 集合 (Set)、有序集合 (Sorted Set)等数据结构，这些数据结构支持范围查询。此外，Redis



支持位图 (Bitmap) 、Hyperloglog 、 地理信息索引 (Geospatial Index) 等数据结构，这些 数据结构支持基于半径的范围查询 (Radius Query)。

Redis是一个高性能的Key-Value 数据库。Redis 支持流水线(/topics/pipelining) 技 术，可以一次发送多个命令，由服务器执行。流水线技术极大地提高了服务器每秒钟可以 处理的请求。根据Redis 开发团队发布的报告 (<http://redis.io/topics/benchmarks>), 在 一台安装2.27GHz Intel Xeon E5520 CPU的服务器上，使用流水线技术和未使用流水线 技术，各类操作的吞吐量如表2-1所示。

**表2-1** **Redis的性能评测结果**

|  |  |
| --- | --- |
| 使用流水线技术 | 未使用流水线技术 |
| SET操作：552028.75 RPS  GET操作：707463.75 RPS  LPUSH操作：767459.75 RPS  LPOP操作：770119.38 RPS | SET操作：122556.53 RPS  GET操作：123601.76 RPS  LPUSH操作：136752.14 RPS  LPOP操作：132424.03 RPS |

注 ：RPS,requests per second, 即每秒钟处理的请求数量。

Redis 数据库除了支持丰富的数据类型和具有极高性能之外，还具有如下技术特色， 包括支持数据复制、支持 Lua 脚本语言、使用LRU(Least Recently Used, 最近最少使 用)淘汰策略、支持事务处理 (Transaction), 以及支持数据的磁盘持久化等。

Redis 数据库通过Redis Sentinel组件，实现系统的高可用性 (High Availability) 。Senti- nel 组件实现了对主从节点的持续监控，并且把错误情况通过 API 通知管理员或者某个计算 机程序.在主节点失败的情况下自动把某个节点提升为主节点。Sentinel 是客户端程序寻找 数据库服务的权威来源。即客户端程序通过连接到Sentinel, 获得当前 Redis 主节点的信息。

Redis Cluster组件是Redis 的集群扩展插件，它可以把数据自动地分割到多个 Redis 节点上。Cluster 组件支持在部分节点失败、无法与之通信的情况下，整个集群仍然可以 继续提供持续的服务，即 Redis的设计目标是在网络分割的情况下，仍然提供一定的系统 的可用性。读者可以回顾一下上文论述的CAP 理论，显然 Redis 的设计原则突出支持 A (Availability) 和 P(Network Partition Tolerance), 牺牲了一致性 (Consistency)。

Redis 在某些应用中可以承担后台数据库的角色，也可以在整个系统中作为一个高速 缓存 (Cache) 或者消息代理 (Message Broker) 来使用。

**2.2.4 Column Family 数** **据** **库**

**1.Big Table**

Big Table 是 Google 基于GFS(Google File System) 分布式文件系统、CLS(Chub-

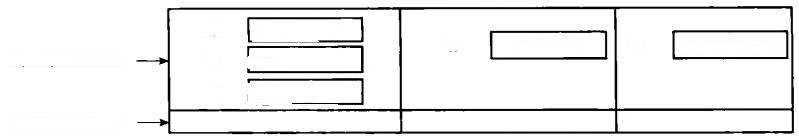
by Lock Service) 分布式加锁服务开发的，大型分布式NoSQL 数据库系统。它支持行、 列、时间戳索引，支持PB 级海量数据的处理，运行于廉价集群.易于扩展，支持动态伸 缩。Big Table 支持大量并发的读操作，同时支持数据更新。和 Dynamo 一样，Big Table 的设计思路极大影响了后续各个NoSQL 数据库系统的研发。

Big Table 数据库系统的存储结构是典型的Column Family 存储，它同样通过键值对 基础数据模型对数据进行建模，但是Value 具有更精巧的结构，即一个 Value 包含多个 列，这些列还可以分组 (Column Family), 呈现出嵌套映射 (Map) 的数据结构特点。由 于每列数据是带有时间戳 (Timestamp) 的，可以在Column Family的每个Column 里维

护多个 Value 版本。在需要对历史数据的变动情况进行分析的场合，这样的建模方法能够 提供有力的支持。

图2 - 7展示了 一 个 Big Table 表格的实例。反转的 URL([www.cnn.com](https://www.cnn.com) 反转为 com.cnn.www) 作为表格的 key, 这个表格有两个Column Family, 分别是Contents 和 Anchor 。Contents 保留了页面的内容，每个时间戳对应一个页面的内容，可以保留该页 面的不同历史版本。Anchor 则保存了指向这个页面(即引用该页面)的其他页面的锚点 (即其他页面的超链接指向本页面)的文本信息。在这个实例里， CNN 的主页被 Sports Illustrated(cnnsi.com) 和 My Look(my.look.ca) 两个页面指向，因此每行记录有两列 anchor:cnnsi.com 和 anchor:my.look.ca, 它们隶属于同一个Column Family。

Contents列 anchor:cnnsi.com 列 anchor:my.look.ca 列



<html> … …

t8 "CNN.com"

<html> … …

<html … …

com.abc.www

t3 t5

t6

**com.cnn.www**

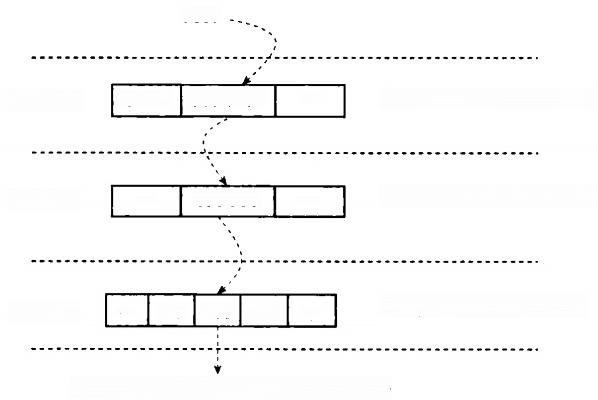
"CNN"

t9

**图** **2** **-** **7** **Big Table 数据库的数据模型**

Big Table 系统采用Master/Slave 的主从系统架构。Master 节点负责保存元信息，并 且实时监控每个数据块(称为Tablet) 的大小、负载情况、服务器的可用性等，对用户的 查询进行路由选择，选择合适的Tablet Server 进行服务。Tablet Server 负责数据块的管 理，数据是按照 Key 进行排序的，每个Tablet 服务器负责一部分数据的管理。

Big Table使用一种层次性范围 (Range) 分区方法，把数据分割成Tablet, 分布到各 个节点上。两个小的 Tablet 可以合并，大的Tablet 可以再分解，总是保持一个 Tablet 的 大小在100MB～200MB 之间。当元信息变得很大时，可以把元信息进一步划分到多个 Tablet 上，形成了3层导航寻址架构，如图2-8所示。



900?

|  |  |  |
| --- | --- | --- |
| 服务器A | 0-499 500-1500 | 元数据 (META DATA)0 |
| 服务器B | . 850-950 | 元数据 (META DATA)1 |
| 服务器C | 899 900 901 | 实际数据 (REAL DATA) |

Key900对应的数据 (Value for Key 900)

**图** **2** **-** **8** **Big Table 的** **Range 分区技术**

Big Table为Google 的搜索、地图、财经、打印、社交网络、视频共享 (YouTube) 以及博客等业务 ， 提供了底层数据存储和操作的支持 。

**2.HBase**

HBase 是受 Big Table启发而开发的基于Column Family 存储模型的开源NoSQL 数 据库，它是整个 Hadoop 生态系统的重要组成部分。在接口方面，HBase 提供传统的SQL 查询接口，可以方便地对数据进行增加、删除、修改、查询和简单汇总(聚集)。

HBase 凭借其强大的扩展能力，应用于日志处理等领域(比如电信应用的日志分析)。 Facebook 对 HBase 进行了持续的改进，极大地提高了其吞吐能力(尤其是写入能力),达 到每天完成200亿个写操作(折合每秒23万个写操作)的性能，对社交网络用户的交互 行为进行记录和分析。

**2.2.5 Document 数** **据** **库**

基于 Document (文档)存储模型的数据库技术由来已久，比如IBM 的 Lotus Notes 等。这里介绍的基于Document 存储的NoSQL 技术，是传统文档数据库技术的新发展。 Document 数据库技术仍然以键值对存储模型作为基础模型。这个模型可以对文档的历史 版本进行追踪，每个文档是一个<Key,Value> 的列表，每个 Value 还可以是一个 <Key,Value> 列表，形成循环嵌套的结构。文档格式一般采用JSON(JavaScript Ob- ject Notation), 或者类似于JSON 的格式。

JSON 是一种轻量级的、基于文本的且独立于语言的数据交换格式，它比 XML 更轻 巧，是XML 数据交换的一个替代方案。假设有一个employee 对象，有姓、名、员工编 号、头衔等信息，使用JSON 表示的具体形式如下：

|  |
| --- |
| employee:  firstName:"John",  lastName:"Doe",  employeeNumber:123, title:"Accountant"  一  } |

对于某些特定的应用来讲，Document 存储的效率更高。Document 存储给予数据库设 计者极大的灵活性对数据进行建模，但是对数据进行操作的编程负担落在了程序员身上。 由于数据的循环嵌套结构特点，应用程序有可能变得更加复杂，难以理解和维护.需要掌 握好灵活性和复杂性之间的平衡。

**MongoDB**

MongoDB 是一款分布式文档数据库，它为大数据量、高度并发访问、弱一致性要求 的应用而设计。MongoDB 具有高度的扩展性能.在高负载的情况下，可以通过添加更多 的节点，保证系统的查询性能和吞吐能力。

MongoDB 本质上是 Key-Value 数据库，它是模式自由的 (Schema Free), 也就是存 储在 MongoDB 数据库中的文件，我们无须定义它的结构模式。如果需要的话，可以把不 同结构、不同类型的文档保存到MongoDB 数据库中。

文档被划分成组，存储在数据库中。一个文档分组称为一个集合 (Collection) 。 每个 集合在数据库中有一个唯一的标识，它可以包含无限数目的文档。集合的概念可以对应到 关系数据库表格 (Table), 文档则可以对应到关系数据库的一条记录 (Record) 。 但是在 这里，无须为集合定义任何模式 (Schema)。

存储在集合中的文档，被存储成键值对的形式，键用于唯一标识一个文档，为字符串 类型，值则可以是任何格式的文件类型，包括二进制JSON 形式BSON(Binary JSON)。 通过二进制数据存储，可以管理大型的音频、视频等多媒体数据对象。

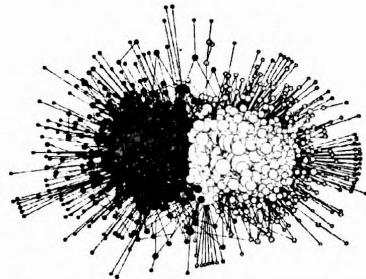
MongoDB 数据库支持增加、删除、修改、简单查询等主要的数据操作以及动态查询， 并且可以在复杂属性上建立索引，当查询包含该属性的条件时，可以利用索引获得更高的 查询性能。MongoDB 提供Ruby,Python,Java,C+ 十 ，PHP 等多种语言的编程接口， 方便用户使用不同语言编写客户端程序，连接到数据库进行数据操作。为了支持业务的持 续 性 ，MongoDB 通过复制技术实现节点的故障恢复。

MongoDB 并不能替代关系数据库，因为它缺乏OLTP 的能力(保证ACID) 和性能 . 它主要应用到大规模、低价值的数据存储和管理场合。比如，欧洲原子能中心使用Mon- goDB 来存储大型强子对撞机实验的部分数据。

**2.2.6 Graph 数** **据** **库**

随着社交网络、科学研究(药物、蛋白质研究)以及其他应用领域不断发展的数据管 理需要，更多的数据以图作为基础模型进行表达更为自然。这些数据的数据量是极其庞大 的，比如，Facebook 拥有超过8亿的用户，对这些用户的交互关系进行管理、分析是极 大的挑战。

这些分析不仅仅是根据一定的条件查找图的节点或者图的边，更为复杂的处理是对图 的结构进行分析，比如社区的检测等(如图2-9所示)。



**图2-9** **美国政治相关博客之间超链接的分析(自由派与保守派)**

资料来源：<http://www.nature.com/nphys/journal/v8/nl/full/nphys2162.html.>

**Neo4J**

Neo4J 是 一 个用Java 语言实现的高性能的图数据库，它的基础数据结构是图 (Graph) 而不是二维表。在一个图中，包含两种基本的数据对象，分别是节点 (Node) 和关系 (Relationship) 。 所有的节点通过关系连接起来形成网络结构 (Network) 。 节点和 关系可以包含键值对(<Key,Value>) 形式的属性。

Neo4J 是模式自由的 (Schema Free),即节点可以表达各种不同的对象，关系可以表 达现实世界中各种对象之间的联系。Neo4J 使用一套易于学习的查询语言Cypher, 支持图 数据的增加、删除、修改、查询等操作。经过精心的数据结构设计和操作算法优化，它达 到2.7秒之内完成100万节点的遍历的极高性能，使得Neo4J 成为支持大规模图数据管理 和查询的数据库引擎。为了把 Neo4J 的应用扩展到企业应用场景，Neo4J 提供了兼容 ACID 语义要求的事务操作能力，并且通过主从复制、联机备份等技术，实现系统的高可 用性。

Neo4J 可以作为嵌入式数据库使用，也可以作为单独的服务器使用。在后一种应用场 景下，它提供了Rest(Representation State Transfer) 接口，用户可以使用PHP,.NET

和Java Script 等语言进行数据操作，方便应用程序的开发。

Neo4J 的典型应用领域包括语义网和RDF 数据、Linked Open Data、地理信息系统 (GIS) 、 基因分析、社交网络、推荐等，甚至在传统RDBMS 的某些应用领域，也有它的 用武之地。比如一些适合用图来表达和处理的数据.包括文件夹结构、产品分类、元信息 管理等，以及具体领域的数据，比如金融领域的知识图谱构建和欺诈检测、电信领域的通 话关系分析等。

**2.3 New SQL数据库技术**

NewSQL 是一类新式的关系数据库管理系统的统称。这类系统一般针对OLTP 类工 作负载，提高了系统的扩展能力.使之具有类似NoSQL 系统的扩展性能，但是仍然保持 传统数据库的优势，特别是对 ACID 事务的支持以及对 SQL 查询语言的支持。

NewSQL 系统可以分为三类：(1)通过全新的架构设计和产品实现，在保持ACID 特 性的基础上，支持更高的系统扩展性，比如Google Spanner,VoltDB等。(2)对现有的 开源数据库系统进行深入的优化(特别是存储、索引以及查询模块)所研发的新的 SQL 数据库，比如TokuDB,MemSQL 等。(3)通过中间件软件，支持数据的透明划分，把整 个数据库分割 (Sharding) 在多个节点上运行，获得更高的性能，比如ScaleBase 等。在 这里，我们将介绍 VoltDB 和 Google Spanner 两个典型的NewSQL 系统。

**2.3.1 VoltDB 数据库**

VoltDB 数据库是原型系统H-Store 的商业化版本，它的设计者是数据库界的知名专 家Michael Stonebraker(2014 年图灵奖获得者)。 VoltDB 是一款高性能的 NewSQL 数 据库。

NoSQL 数据库突破了传统关系数据库的扩展性瓶颈，把数据分布到成百上千个节点 上并行处理，但是它损失了数据的一致性。NoSQL 数据库一般不支持完全的ACID 事务 处理，而是采用了最终一致性模型，保证数据的一致性。VoltDB 是一款内存数据库系统， 提供了类似于NoSQL 数据库的扩展性，同时没有放弃传统关系数据库系统支持的ACID。

在一个采用类似 TPC-C 负载的性能评测中，VoltDB 在单个节点配置下，获得了超过 某个传统数据库系统(测试方没有公开产品名称)40倍的吞吐量，达到53000TPS (每秒

执行的事务数， Transactions per Second)。在12个节点组成的数据库集群上 . 获得 560000TPS 的吞吐量。TPC-C 是事务处理性能委员会 (Transaction Processing Perform- ance Council) 定义的面向联机事务处理 (OLTP) 应用的数据库性能评测基准 (Bench- mark)。它模拟了客户购买商品、商家对订单进行处理和货物分发的数据模型和商业流程。 VoltDB 在提供 ACID 事务支持的情况下，获得极高的性能和高度的扩展能力。为此，它 采用了一系列新的设计。

通过分析发现.传统关系数据库系统大概把10%的CPU 时间花在提取和更新记录上， 把90%的时间花在缓冲区管理、加锁 (Locking, 实现事务间的数据操作协调)、闩锁 (Latching, 协调多线程对数据结构的存取，实现数据结构保护)以及日志等操作上。

VoltDB 把数据保存在集群内存中.整个 VoltDB 数据库由若干分区 (Partition) 组 成.这些分区分布在各个站点 (Site, 即服务器)上。每个站点上的数据分区，通过单一 的线程 (Singled Thread) 进行存取，避免了多线程环境下的加锁 (Locking) 和闩锁 (Latching) 操作带来的开销。所有的事务请求被串行执行。

VoltDB 使用SQL 作为查询语言，这是广大开发人员早已熟悉的数据库查询语言。对 数据库的存取通过存储过程 (Stored Procedure) 来实现，意味着用户不能发起即席查询 (Ad-HocQuery. 即用户临时发起的查询)。存储过程是用Java 语言编写的函数.SQL 语 句嵌入在存储过程中，这些存储过程编译成可执行代码，由数据库引擎执行，而不是像普 通的 SQL 查询语句那样解释执行。比起通过JDBC 接口存取数据库来讲 · 这种方式下， 每个事务只需客户端和服务器的一次往返((ne Trip) 传输，避免了因为多次通过网络调 用数据库服务器带来的延迟。每个对存储过程的调用是一个事务。如果事务成功执行.那 么提交，否则进行回滚。虽然SQL 语句需要事先确定，但是可以通过在运行时绑定具体 参数，实现灵活的数据存取。

对数据进行分区时，考虑的首要问题是，尽量使得事务(数据修改和查询)在一个服 务器上完成。对数据分区所使用的 Key 的选择，需要精心考虑，因为某些查询如果没有用 到这个 Key, 它就需要存取多个分区(即多个服务器)。比如我们对部门表和员工表都按 照部门编号进行分区 (Partition). 那么进行这两个表连接操作时，比如查询某个部门年 龄在一定范围的员工，就可以在一个服务器上完成.无须服务器之间的信息交换，因为某 个部门和该部门的员工信息都已经在某个服务器上。如果我们对部门表按照部门编号进行 分区，对员工表按照员工编号进行分区，对于某个部门来讲.它的员工信息分散在各个服 务器上，那么执行上述查询时，需要在服务器之间交换信息。

各个分区并行地各自执行自己的查询，使得系统获得更高的吞吐量。需要指出的是， 根据工作负载的某些查询确定的数据分区策略，可能不能照顾到其他查询、使得它们也能 够在单个服务器上完成，于是存取多个服务器是不可避免的。DBA (数据库管理员)需要 根据数据和查询特点，采用适当的 Key 进行数据分区.使得这样的查询造成的影响尽 量低 。

对于某些数据库表.由于其数据量非常小，而且不轻易进行更新， 一般可以把它复制 到各个服务器上，方便进行数据的连接操作 (Join) 。 比如数据库里有部门表和员工表. 查询涉及这两个表。部门表只有少数的几条记录，数据量很少，不经常改动，那么就可以 把它复制到各个服务器，方便和员工表的连接操作，加快查询。比如查询某个部门年龄在 一定范围的员工，就需要连接部门和员工两张表。

VoltDB 通过快照和日志技术，支持事务的持久性和数据库的可恢复性。在它的日志



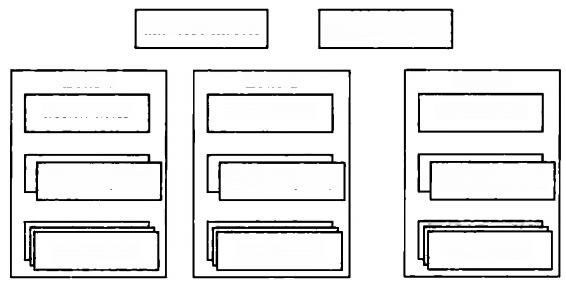
中，并未记录数据改变的情况，而是把对存储过程(具有编号)的调用及其参数，按照调 用的串行顺序记录下来，每个节点保留自己的日志信息，这种日志方法称为命令日志 (Command Logging)。如果对日志进行同步处理 (Synchronous Logging),那么当日志已 经持久化到硬盘以后.存储过程才能提交。用户也可以指定，把若干存储过程调用的日 志，成批记录到磁盘，对事务进行批量提交 (Batch Commit), 从而提高吞吐量，但是会对单 个事务的响应时间造成一定影响。快照 (Snapshot) 是数据库的数据在某个时间点的状态。 VoltDB 可以自动地每隔一定的时间间隔，创建快照并存盘。当数据库关闭以后(包括异常 关闭和正常关闭),可以利用快照和日志信息，把数据库恢复到最近的一致状态。

**2.3.2 Google Spanner 数据库**

Spanner 是 Google 开发的跨越数据中心的、提供全局一致性的关系数据库系统。从用 户角度来看，和传统的关系数据库没有什么两样。本质上， Spanner 是一个高度扩展的、 支持多版本的、支持同步复制的分布式数据库系统。 Spanner 支持丰富的SQL 查询语言功 能，提供极高的查询性能(查询响应时间达到几十毫秒，只读事务的响应时间甚至达到几 个毫秒)。

Spanner 数据库组织为若干区域 (Zone), 每个区域大致可以认为是Big Table的 一个 部署。Universe Master作为一个管理工具.显示所有区域的状态信息.支持交互式的调 试功能等。Placement Driver提供跨越区域数据迁移的管理功能。

Zonemaster 相当于 Big Table 的 Master, 负责管理 Spanserver 上 的 数 据 。Location Proxy 存储数据的位置 (Location) 信息，客户端要先访问Location Proxy, 才能知道数 据在哪个Spanserver 上 。Spanserver 相当于Big Table的 Tablet Server, 用于存储具体的 数据。如图2- 10所示。



**universe master placement driver**

Zone 1 Zone 2 Zone N

zonemaster zonemaster **zonemaster**

location proxy **location proxy location proxy**

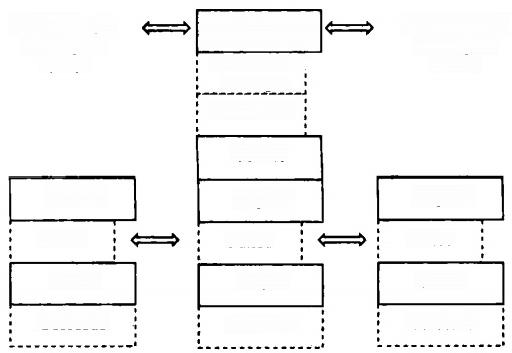
spanserver **spanserver spanserver**

**图2-10** **Spanner** **的服务器组织方式**

在 Big Table中 ，Tablet 只是行数据的容器， Tablet 内部的行的地位是对等的。 Span- ner 对 Tablet 进 一 步进行结构划分，多了 一 层 Dictionary 结 构 。Dictionary 是 一 些 <Key,Value> 的集合， 一个 Dictionary 里面的 Key 有一样的前缀。 Dictionary是数据复 制和数据放置的基本单位，比如某个 Dictionary 需要分布到亚洲和美洲，共有3个副 本等。

Spanserver 的内部设计和 Big Table 系统非常相似，如图2- 11所示。每个数据中心 运行一套Colossus(Google 分布式文件系统， Google File System的新版本 GFS II)。每

台服务器管理100～1000个 Tablet 。 每 个 Tablet 上有 一 个 Paxos 状态机， Paxos 是 一 个 分布式一致性协议，管理 Tablet 的元数据和日志信息(对 Paxos 算法的详细介绍请参考 第 3 章 “OLAP 与结构化数据分析”)。



Particpan leader

Transaction!

manager

Lock table

Leader

Replica

Paxos

Tablet

Colossus

Other group's

Participant

leader

Replica

Paxos

Tablet

Colossus

Other group's

Participant

leader

Replica

Paxos

Tablet

Colossus

Data Center X Data Center Y Data Center Z

**图2-** **11** **Spanserver 内部结构**

Paxos 选出 一个副本 (Replica) 做领导者 (Leader),Leader 的寿命默认为10秒，10

秒后重新选举。 Leader 是数据的主副本 (Master), 其他副本的数据都从主副本那里进行 复制。读请求可以路由到任意的副本，但是写请求都被引导到主副本 Leader 上。这些副 本统称为一个 Paxos 组 (Paxos Group)。

每个主副本 Leader 所在的服务器，实现一个锁表 (Lock Table), 负责管理并发操作。 锁表记录了两阶段提交(关于两阶段提交协议的详细信息，请参考第3章 “OLAP 与结构 化数据分析”)需要的锁信息。每个主副本 Leader 所在的Spanserver 上还有一个事务管理 器 (Transaction Manager)。如果事务在 一个 Paxos Group 里面，可以绕过事务管理器对 事务进行协调运行。 一旦事务跨多个 Paxos Group. 就需要事务管理器来协调，其中一个 Transaction Manager 被选为事务协调者，协调所有的事务参与者，保证事务的正确执行。

Spanner 可以指定数据分布的位置。把数据指定到离用户较近的地方，可以减少读延 迟。指定各个数据副本(保证系统的容错)之间有多远，可以控制写的延迟 . 因为写延迟 取决于最远的副本。

在数据模型方面， Spanner 的 行 模 型 是<Key:string,Timestamp:int64>→Row

Value 。 在这个模型中，强化了行 (Row) 的概念，不再突出列 (Column) 的概念，其中

的时间戳 (Timestamp) 是赋予整行的，使得 Spanner 可以实现多版本并发操作。

Spanner 支持读写的外部一致性，以及基于时间戳的全局的读一致性。为了实现这两 个特性， Spanner 使用了全球时间同步机制，可以在数据提交时给出其时间戳。由于时间 是序列化的，所以系统具有外部一致性。全球时间同步机制通过具有GPS 和原子钟功能 的 True Time 编程接口 (API) 实现。基于这两个特性， Spanner 可以实现 一致的备份功 能(即备份里只有提交的数据，没有未提交的脏数据),以及对 Schema 进行原子的 (A- tomic) 修改。

在 True Time的 基 础 上 ，Spanner 实现了并发控制，提供外部一致性，它支持四种事 务类型，分别是读写事务、只读事务、客户端提供时间戳的快照读，以及客户端提供时间

范围的快照读等。当某个读写事务是在时间T₁ 写入新的数据，那么其并发控制机制保证 在全球任何地方，指定时间戳为T₁ 的快照读都能够读到刚刚写入的值。时间戳的设计大 大提高了只读事务的性能，只读事务利用系统时间戳读取数据的相应副本，不会对读写事 务发生堵塞作用。对于快照读事务，当用户指定了时间戳或者时间范围， Spanner 会寻找 已经充分更新的副本读取数据。表2-2列出了Spanner 支持的事务类型。

**表2-2** **Spanner的事务处理**

|  |  |  |
| --- | --- | --- |
| 事务类型 | 并发控制方法 | 涉及的副本 |
| Read Write Transaction | 悲观并发控制方法(Pessimistic) | Leader |
| Read Only Transaction | 无须加锁(Lock free) | Leader for timestamp Any other for read |
| Snapshot Read,Client Provided Timestamp | 无须加锁(Lock Free) | Leader or any other |
| Snapshot Read,Client Provided Bound | 无须加锁(Lock Free) | Leader or any other |

**2.4** **思考题**

(1)关系模型、关系操作。

(2)ACID 事务特性。

(3)数据库安全。

(4)并行数据库、分布式数据库。

(5)CAP 理论。

(6)NoSQL 数据库，NoSQL 数据库的四种类型以及典型代表。

(7)NewSQL 数据库，NewSQL 数据库的分类与典型代表。

(8) 一 致性Hash 技术与实例。

(9)Quorum 技术与实例。

(10)向量时钟技术与实例。