第6章: 物理数据库设计

Physical Database Design

邹兆年

哈尔滨工业大学
计算机科学与技术学院
海量数据计算研究中心
电子邮件: znzou@hit.Ericu.cn

2020年春

教学内容[1]

1. 物理数据库设计概述
2. 索引的设计
3. 物理存储结构的设计

[1]课件更新于2020年3月10日

6.1 物理数据库设计概述

Introduction to Physical Database Design

物理数据库设计概述

- 任务: 在逻辑数据库设计的基础上，为每个关系模式选择合适的存储结构和存取方法，使得数据库上的事务能够高效率地运行
- 设计步骤:
  1. 分析影响物理数据库设计的因素
  2. 为关系模式选择存取方法
  3. 设计关系、索引等数据库文件的物理存储结构

数据库负载(Workload)

- 查询列表
  - 查询语句
  - 出现频率
  - 性能要求
- 更新列表
  - 更新语句
  - 出现频率
  - 性能要求

设计步骤1: 分析数据库负载

- 对于负载中的查询，需掌握以下信息
  - 查询涉及的关系
  - 投影属性
  - 选择条件和连接条件涉及的属性
  - 选择条件和连接条件的选择度(selectivity)
- 对于负载中的更新，需掌握以下信息
  - 被更新的关系
  - 更新的类型(增加、删除、修改)
  - 被更新关系上被更新的属性
  - 更新条件涉及的关系
  - 选择条件和连接条件涉及的属性
  - 选择条件和连接条件的选择度

设计步骤2: 选择关系数据库的存取方法

- 存取方法(access methods): DBMS访问关系中元组的方法
  - 顺序访问
  - 索引访问
- 索引(index)是提高查询效率的重要手段之一
  - 是否建立索引?
  - 将哪些属性作为索引的搜索键?
  - 建立什么类型的索引?
  - 使用哪种索引结构?
  - 索引更新代价如何?

设计步骤3: 设计关系数据库的物理存储结构

- 确定如何在存储器上存储关系及索引，使得存储空间利用率最大化，数据操作产生的系统开销最小化
- RDBMS使用的物理存储结构取决于存储引擎，用户可以选择合适的存储引擎，但基本不能改变存储引擎使用的物理存储结构

6.2 索引的设计

Designing Indexes

索引(Index)

- 索引(index)能够帮助DBMS快速找到关系中满足搜索条件的元组
- 索引对于提高查询处理效率至关重要

Example (索引)

索引

Sname	元组地址
Abby	$addr_3$
Ed	$addr_2$
Elsa	$addr_1$
Nick	$addr_4$

Student关系

地址	Sno	Sname	Ssex	Sage	Sdept
$addr_1$	CS-001	Elsa	F	19	CS
$addr_2$	CS-002	Ed	M	19	CS
$addr_3$	MA-001	Abby	F	18	Math
$addr_4$	PH-001	Nick	M	20	Physics

查询: `SELECT Sdept FROM Student WHERE Sname = 'Elsa';`

- 如果没有索引，则只能通过扫描Student关系来完成查询
- 如果有上述索引，则可以通过该索引来快速完成查询

索引的构成

- 索引键(index key): 索引根据一组属性(索引键)来定位元组
- 索引记录了元组的索引键值与元组地址的对应关系
- 索引项(index entry): 索引中的(键值，地址)对
- 索引中的索引项按索引键值排序

Example (索引)

索引

Sname	元组地址
Abby	$addr_3$
Ed	$addr_2$
Elsa	$addr_1$
Nick	$addr_4$

Student关系

地址	Sno	Sname	Ssex	Sage	Sdept
$addr_1$	CS-001	Elsa	F	19	CS
$addr_2$	CS-002	Ed	M	19	CS
$addr_3$	MA-001	Abby	F	18	Math
$addr_4$	PH-001	Nick	M	20	Physics

索引的分类

- 主索引(primary index): 索引键是关系的主键
- 二级索引(secondary index): 索引键不是关系的主键

Example (主索引vs 二级索引)

主索引

Sno	元组地址
CS-001	$addr_2$
CS-002	$addr_3$
MA-001	$addr_4$
PH-001	$addr_1$

二级索引

Sdept	元组地址
CS	$addr_3$
CS	$addr_2$
Math	$addr_4$
Physics	$addr_1$

Student关系

地址	Sno	Sname	Ssex	Sage	Sdept
$addr_1$	CS-001	Elsa	F	19	CS
$addr_2$	CS-002	Ed	M	19	CS
$addr_3$	MA-001	Abby	F	18	Math
$addr_4$	PH-001	Nick	M	20	Physics

- 只能有一个主索引
- 可以有多个二级索引

索引的分类(续)

- 唯一索引(unique index): 索引键值不重复
- 非唯一索引(non-unique index): 索引键值可重复

Example (唯一索引vs 非唯一索引)

唯一索引

Sno	元组地址
CS-001	$addr_2$
CS-002	$addr_3$
MA-001	$addr_4$
PH-001	$addr_1$

非唯一索引

Sdept	元组地址
CS	$addr_3$
CS	$addr_2$
Math	$addr_4$
Physics	$addr_1$

Student关系

地址	Sno	Sname	Ssex	Sage	Sdept
$addr_1$	CS-001	Elsa	F	19	CS
$addr_2$	CS-002	Ed	M	19	CS
$addr_3$	MA-001	Abby	F	18	Math
$addr_4$	PH-001	Nick	M	20	Physics

索引的分类(续)

- 索引中通常存储元组地址
- 根据元组地址读取元组需要进行1次I/O
- 为了节省I/O，可在索引中直接存储元组
- 聚簇索引(clustered index)/索引组织表(index-organized table): 索引中存储的是元组本身
- 非聚簇索引(non-clustered index): 索引中存储的是元组地址
- 一个关系上只能有一个聚簇索引(为什么?)

Example (聚簇索引)

聚簇索引

Sno	Sname	Ssex	Sage	Sdept
CS-001	Elsa	F	19	CS
CS-002	Ed	M	19	CS
MA-001	Abby	F	18	Math
PH-001	Nick	M	20	Physics

MySQL中的索引[2]

- 主索引采用聚簇索引
- 二级索引采用非聚簇索引
- 二级索引中存储的不是元组地址，而是元组的主键值 (好处？坏处?)

Example (MySQL中的主索引和二级索引)

二级索引

Sname	Sno
Abby	MA-001
Ed	CS-002
Elsa	CS-001
Nick	PH-001

主索引

Sno	Sname	Ssex	Sage	Sdept
CS-001	Elsa	F	19	CS
CS-002	Ed	M	19	CS
MA-001	Abby	F	18	Math
PH-001	Nick	M	20	Physics

[2]MySQL InnoDB存储引擎

创建主索引

- 在CREATE TABLE或ALTER TABLE语句中使用PRIMARY KEY声明主键时，自动建立主索引
- 用ASC或DESC声明主索引中的属性按升序还是降序排列
- 只能在CREATE TABLE或ALTER TABLE语句中声明主索引
- 在MySQL中，主索引的名称就是PRIMARY

Example (创建主索引)

```
CREATE TABLE Student (
Sno CHAR(6),
Sname VARCHAR(10),
Ssex ENUM('M', 'F'),
Sage INT,
Sdept VARCHAR(20),
PRIMIARY KEY (Sno ASC));
```

创建二级索引

- 在CREATE TABLE或ALTER TABLE语句中声明二级索引
  KEY|INDEX (索引键) [索引名]
- 用ASC或DESC声明索引属性的排序方式

Example (创建二级索引)

```
CREATE TABLE Student (
Sno CHAR(6) PRIMARY KEY,
Sname VARCHAR(10),
Ssex ENUM('M', 'F'),
Sage INT,
Sdept VARCHAR(20),
KEY (Sname, Sage DESC) key_sname_sage);
```


创建唯一索引

- 在CREATE TABLE或ALTER TABLE语句中声明唯一索引
  UNIQUE [KEY|INDEX] (索引键) [索引名]
- 用ASC或DESC声明索引属性的排序方式

Example (创建唯一索引)

```
CREATE TABLE Student (
Sno CHAR(6) PRIMARY KEY,
Sname VARCHAR(10),
Ssex ENUM('M', 'F'),
Sage INT,
Sdept VARCHAR(20),
UNIQUE (Sname) uk_sname);
```

创建外键索引

- 在CREATE TABLE或ALTER TABLE语句中使用FOREIGN KEY声明外键时，会为外键创建索引
- 建外键索引是为了加快参照完整性检查

Example (创建外键索引)

```
CREATE TABLE SC (
Sno CHAR(6),
Cno CHAR(4),
Grade INT,
PRIMARY KEY (Sno, Cno),
FOREIGN KEY (Sno) REFERENCES Student(Sno));
```

2个外键

创建索引(续)

- 尽量在CREATE TABLE语句中将一个关系上的所有索引都建好
- 可以使用ALTER TABLE语句添加索引
- 除主索引外，可以使用CREATE INDEX语句添加索引

```
CREATE [UNIQUE] INDEX 索引名
ON 关系名 (索引键);
```

删除索引

删除二级索引

- 语句: DROP INDEX 索引名 ON 关系名;
- 删除二级索引不需要重新组织关系中的元组

删除主索引

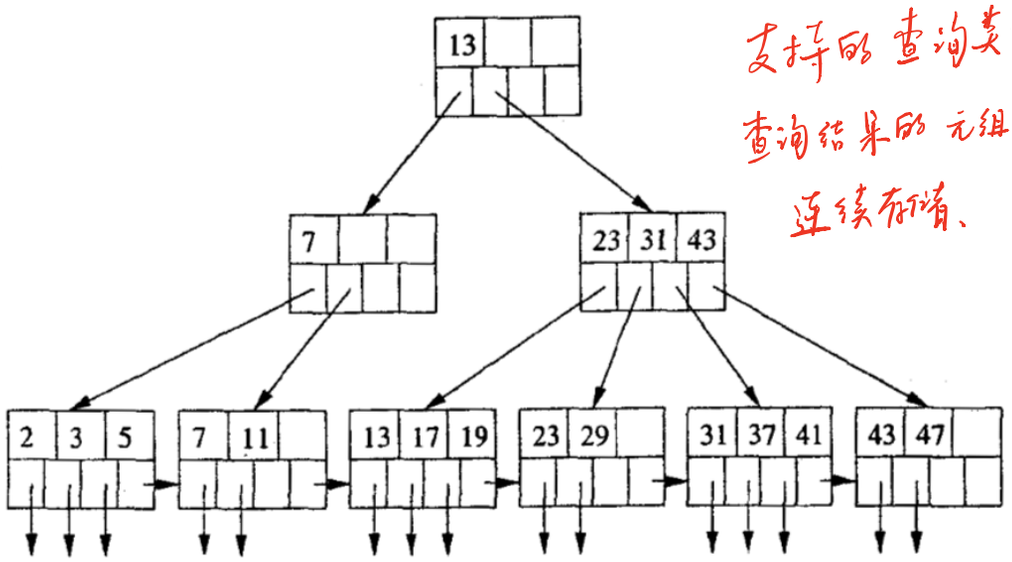
- 语句: DROP INDEX 'PRIMARY' ON 关系名;
- 删除主索引需要重新组织关系中元组
- 删除主索引的代价比删除二级索引的代价高

索引数据结构

- 索引可以用多种数据结构实现
  - B+树
  - 哈希表
  - 跳表(skiplist)
- 不同的索引结构具有不同的功能和特性

B+树(B+ Trees)

- B+树是大多数RDBMS所使用的索引结构
- B+树是一棵平衡多叉树，所有叶节点的深度都相同
- 索引项全部存储在B+树的叶节点中，并按索引键值排序存储



支持的查询类型必须使查询结果的元组的索引键连续存储。

B+树索引支持的查询类型 I

对于B+树索引支持的查询，其查询结果中元组的索引键值在B+树的叶节点中连续存储

Example (B+树索引)

```
CREATE INDEX idx
ON Student (Sname, Sage, Ssex)
USING BTREE;
```

Sname	Sage	Ssex
Cindy	19	F
Ed	18	M
Elsa	19	F
Elsa	19	M
Elsa	20	F
Fawn	18	F

1. 全值匹配: 和所有索引属性进行匹配

```
SELECT * FROM Student
WHERE Sname = 'Elsa' AND Sage = 19 AND Ssex = 'F';
```

2. 匹配最左前缀: 和最前面几个索引属性进行匹配

```
SELECT * FROM Student WHERE Sname = 'Elsa' AND Sage = 19;
```

B+树索引支持的查询类型 II

3. 匹配属性前缀: 只匹配某属性的开头部分

```
SELECT * FROM Student WHERE Sname LIKE 'E%';
```

4. 范围匹配: 在给定范围内对某属性进行匹配

```
SELECT * FROM Student
WHERE Sname BETWEEN 'Ed' AND 'Emma';
```

5. 精确匹配某一属性并范围匹配另一属性

```
SELECT * FROM Student
WHERE Sname = 'Elsa' AND Sage BETWEEN 18 AND 20;
```

6. B+树索引还支持按索引属性进行排序

Sname	Sage	Ssex
Cindy	19	F
Ed	18	M
Elsa	19	F
Elsa	19	M
Elsa	20	F
Fawn	18	F

B+树索引的限制 I

如果查询结果中元组的索引键值在B+树的叶节点中不连续存储，则B+树不支持该查询

Example (B+树索引)

```
CREATE INDEX idx
ON Student (Sname, Sage, Ssex)
USING BTREE;
```

Sname	Sage	Ssex
Cindy	19	F
Ed	18	M
Elsa	19	F
Elsa	19	M
Elsa	20	F
Fawn	18	F

1. 必须从索引的最左属性开始查找

```
SELECT * FROM Student WHERE Sage = 19;
```

不能在该索引上执行这个查询

2. 条件中不能包含表达式

可能不是连续存储
因此不行

B+树索引的限制 II

```
SELECT * FROM Student
WHERE Sname = 'Elsa' AND 2020 - Sage = 2000;
```

在索引上只能根据条件Sname = 'Elsa'进行查找，在返回的元组上验证条件2020 - Sage = 2000

3. 不能跳过索引中的属性

```
SELECT * FROM Student
WHERE Sname = 'Elsa' AND Ssex = 'F';
```

在索引上只能根据条件Sname = 'Elsa'进行查找，在返回的元组上验证条件Ssex = 'F'

4. 如果查询中有关于某个属性的范围查询，则其右边所有属性都无法使用索引查找

```
SELECT * FROM Student WHERE Sname = 'Elsa'
AND Sage BETWEEN 18 AND 20 AND Ssex = 'F';
```

在索引上只能根据条件Sname = 'Elsa' AND Sage BETWEEN 18 AND 20进行查找，在返回的元组上验证条件Ssex = 'F'

哈希索引(Hash Index)

- 哈希索引是基于哈希表(hash table)实现的[3]
- 哈希表中的索引项是(索引键值的哈希值, 元组地址)
- 哈希索引只支持对所有索引属性的精确匹配

Example (哈希索引)

哈希索引

h(Sname)	地址
111	$addr_2$
222	$addr_4$
333	$addr_3$
444	$addr_1$

Student关系

地址	Sno	Sname	Ssex	Sage	Sdept
$addr_1$	CS-001	Elsa	F	19	CS
$addr_2$	CS-002	Ed	M	19	CS
$addr_3$	MA-001	Abby	F	18	Math
$addr_4$	PH-001	Nick	M	20	Physics

- h('Abby') = 333
- h('Ed') = 111
- h('Elsa') = 444
- h('Nick') = 222

[3]MySQL中只有MEMORY存储引擎支持哈希索引，创建哈希索引时使用USING HASH

哈希索引的限制

根源是 HASH 函数的性质

```
CREATE INDEX idx ON Student (Sname, Sage, Sex) USING HASH;
```

- 哈希索引不支持部分索引属性匹配

  SELECT * FROM Student WHERE Sage = 19; (不能使用索引)
- 哈希索引只支持等值比较查询(=, IN)，不支持范围查询

  SELECT * FROM Student WHERE Sname = 'Elsa' AND Sage < 19 AND Ssex = 'F'; (不能使用索引)
- 哈希索引并不是按照索引值排序存储的，所以无法用于排序
- 哈希索引存在冲突问题 → HASH 碰撞，但是概率低

给出 X_1 和 X_2 若 $H(X_1)=H(X_2)$ 可认为 $X_1=X_2$.

索引设计的过程

1. 分析工作负载
   - 分析SELECT语句和UPDATE语句涉及哪些关系
   - 分析WHERE子句、连接条件和GROUP BY子句中涉及哪些属性
2. 依次检查每个重要的查询
   - 确定查询优化器为该查询制定的执行计划
   - 判断能否通过增加新的索引获得更高效的查询计划
     - ★ 需要了解索引技术(第7章)
     - ★ 需要了解查询优化技术(第9章)
   - 如果可以，将新的索引作为候选
3. 权衡索引更新代价

一级索引代价 >> 二级索引代价

设计技巧1: 伪哈希索引(Pseudo-Hash-Index)

- 尽管有些存储引擎不支持哈希索引，但我们可以模拟哈希索引

Example (伪哈希索引)

在Student关系上创建Sname属性上的伪哈希索引

1. 在Student中增加SnameHash属性，存储Sname属性的哈希值CRC32(Sname)
2. 删除Sname上的索引，创建SnameHash上的索引
3. 在查询时，对查询语句进行修改

```
SELECT * FROM Student
WHERE Sname = 'Elsa' AND SnameHash = CRC32('Elsa');
```

伪哈希索引

S.H.	地址
111	$addr_2$
222	$addr_4$
333	$addr_3$
444	$addr_1$

Student关系

地址	Sno	Sname	Ssex	Sage	Sdept	S.H.
$addr_1$	CS-001	Elsa	F	19	CS	444
$addr_2$	CS-002	Ed	M	19	CS	111
$addr_3$	MA-001	Abby	F	18	Math	333
$addr_4$	PH-001	Nick	M	20	Physics	222

设计技巧1: 伪哈希索引(续)

- 优点: 查询速度快
- 缺点:
  - 仅支持等值查询，不支持范围查询
  - 需要改写查询
  - 需要在数据更新时维护哈希值属性

Example (伪哈希索引)

伪哈希索引

S.H.	地址
111	$addr_2$
222	$addr_4$
333	$addr_3$
444	$addr_1$

Student关系

地址	Sno	Sname	Ssex	Sage	Sdept	S.H.
$addr_1$	CS-001	Elsa	F	19	CS	444
$addr_2$	CS-002	Ed	M	19	CS	111
$addr_3$	MA-001	Abby	F	18	Math	333
$addr_4$	PH-001	Nick	M	20	Physics	222

设计技巧2: 前缀索引(Prefix Index)

- 当索引很长的字符串时，索引会变得很大，而且很慢
- 当字符串的前缀(prefix)具有较好的选择性时，可以只索引字符串的前缀
  - 例: 'E', 'El', 'Els'都是'Elsa'的前缀

Definition (索引的选择性(selectivity))

不重复的索引键值和元组数量N的比值，即$\frac{\texttt{COUNT(DISTINCT A)}}{\texttt{COUNT(*)}}$，范围在$1/N$和1之间。选择性越高，索引的过滤能力越强。

Example (前缀索引，前缀长度= 1，选择性= 0.75)

```
CREATE INDEX idx1 ON Student (Sname(1));
```

前缀索引idx1

Sname(1)	地址
A	$addr_3$
E	$addr_1$
E	$addr_2$
N	$addr_4$

Student

地址	Sno	Sname	Ssex	Sage	Sdept
$addr_1$	CS-001	Elsa	F	19	CS
$addr_2$	CS-002	Ed	M	19	CS
$addr_3$	MA-001	Abby	F	18	Math
$addr_4$	PH-001	Nick	M	20	Physics

设计技巧2: 前缀索引(续)

Example (前缀索引，前缀长度= 2，选择性= 1 $\star$)

```
CREATE INDEX idx2 ON Student (Sname(2));
```

前缀索引idx2

Sname(2)	地址
Ab	$addr_3$
Ed	$addr_2$
El	$addr_1$
Ni	$addr_4$

Student

地址	Sno	Sname	Ssex	Sage	Sdept
$addr_1$	CS-001	Elsa	F	19	CS
$addr_2$	CS-002	Ed	M	19	CS
$addr_3$	MA-001	Abby	F	18	Math
$addr_4$	PH-001	Nick	M	20	Physics

Example (前缀索引，前缀长度= 3，选择性= 1)

```
CREATE INDEX idx3 ON Student (Sname(3));
```

前缀索引idx3

Sname(3)	地址
Abb	$addr_3$
Ed	$addr_2$
Els	$addr_1$
Nic	$addr_4$

Student

地址	Sno	Sname	Ssex	Sage	Sdept
$addr_1$	CS-001	Elsa	F	19	CS
$addr_2$	CS-002	Ed	M	19	CS
$addr_3$	MA-001	Abby	F	18	Math
$addr_4$	PH-001	Nick	M	20	Physics

设计技巧2: 前缀索引(续)

前缀索引的缺点

- 前缀索引不支持排序(ORDER BY)
- 前缀索引不支持分组查询(GROUP BY)

排序 Ed 在 Elsa 前，这对于前缀1分组也做不到。
1分组需要 Elsa，Ed 是不同组，这从前缀索引中无法做到

Example (前缀索引，前缀长度= 1，选择性= 0.75)

```
CREATE INDEX idx1 ON Student (Sname(1));
```

前缀索引idx1

Sname(1)	地址
A	$addr_3$
E	$addr_1$
E	$addr_2$
N	$addr_4$

Student

地址	Sno	Sname	Ssex	Sage	Sdept
$addr_1$	CS-001	Elsa	F	19	CS
$addr_2$	CS-002	Ed	M	19	CS
$addr_3$	MA-001	Abby	F	18	Math
$addr_4$	PH-001	Nick	M	20	Physics

Question

*Sno*属性上适合建前缀索引吗？有好办法吗？

设计技巧3: 单个多属性索引 vs. 多个单属性索引

```
SELECT * FROM Student
WHERE Sname = 'Elsa' AND Sage = 19 AND Ssex = 'F';
```

单个多属性索引

```
CREATE INDEX idx ON Student (Sname, Sage, Sex);
```

该索引能直接支持上面的查询

多个单属性索引

```
CREATE INDEX idx1 ON Student (Sname);
CREATE INDEX idx2 ON Student (Sage);
CREATE INDEX idx3 ON Student (Sex);
```

1. 在idx1上查找Sname = 'Elsa'的元组地址
2. 在idx2上查找Sage = 19的元组地址
3. 在idx3上查找Ssex = 'F'的元组地址
4. 索引合并(index merge): 对上述3个元组地址集合取交集，再取元组

设计技巧3: 单个多属性索引 vs. 多个单属性索引(续)

多个单属性索引的缺点

- 效率没有在单个多属性索引上做查询高
- 索引合并涉及排序，要消耗大量计算和存储资源
- 在查询优化时，索引合并的代价并不被计入，故“低估”了查询代价

索引合并：慢

设计技巧4: 索引属性的顺序

在属性Sname, Sage, Ssex上可以建立6种不同顺序的索引，哪种好?

1. `CREATE INDEX idx1 ON Student (Sname, Sage, Sex);`
2. `CREATE INDEX idx2 ON Student (Sname, Ssex, Sage);`
3. `CREATE INDEX idx3 ON Student (Sage, Sname, Sex);`
4. `CREATE INDEX idx4 ON Student (Sage, Ssex, Sname);`
5. `CREATE INDEX idx5 ON Student (Ssex, Sname, Sage);`
6. `CREATE INDEX idx6 ON Student (Ssex, Sage, Sname);`

经验法则

当不考虑排序(ORDER BY)和分组(GROUP BY)时，将选择性最高的属性放在最前面通常是好的，可以更快地过滤掉不满足条件的记录

设计技巧5: 聚簇索引

优点

- 相关数据保存在一起，可以减少磁盘I/O
- 无需“回表”，数据访问更快

缺点

- 设计聚簇索引是为了提高I/O密集型应用的性能，如果数据全部在内存中，那么聚簇索引就没什么优势了
- 聚簇索引上元组插入的速度严重依赖于元组的插入顺序
- 更新聚簇索引键值的代价很高，需要将每个被更新的元组移动到新的位置
- 如果每条元组都很大，需要占用更多的存储空间，全表扫描变慢

Example (聚簇索引)

聚簇索引

Sno	Sname	Ssex	Sage	Sdept
CS-001	Elsa	F	19	CS
CS-002	Ed	M	19	CS
MA-001	Abby	F	18	Math
MA-002	Cindy	F	19	Math
PH-001	Nick	M	20	Physics

设计技巧5: 聚簇索引(续)

主键上建立聚簇索引

```
CREATE TABLE Student (
Sno CHAR(16) PRIMARY KEY,
Sname VARCHAR(10),
Ssex ENUM('M', 'F'),
Sage INT,
Sdept VARCHAR(20));
```

- 优点: 元组按Sno属性聚集存储
- 缺点:
  - 若不按Sno递增顺序插入元组，速度会很慢
  - 二级索引的索引项中存储Sno的值，导致二级索引很大

Example (主键上的聚簇索引)

二级索引

Sname	主键值
Abby	MA-001
Ed	CS-002
Elsa	CS-001
Nick	PH-001

聚簇索引

Sno	Sname	Ssex	Sage	Sdept
CS-001	Elsa	F	19	CS
CS-002	Ed	M	19	CS
MA-001	Abby	F	18	Math
PH-001	Nick	M	20	Physics

设计技巧5: 聚簇索引(续)

代理键(surrogate key)上建立聚簇索引

```
CREATE TABLE Student (
Sno CHAR(16),
Sname VARCHAR(10),
Ssex ENUM('M', 'F'),
Sage INT,
Sdept VARCHAR(20),
ID INT AUTO_INCREMENT,
PRIMARY KEY (ID));
```

- 主键ID与应用无关，称为代理键
- 优点:
  - 一定按ID递增顺序插入记录，插入速度快
  - 二级索引的索引项中存储ID的值，二级索引更小
- 缺点: 记录不按Sno属性聚集存储

Example (代理键上的聚簇索引)

例: CS的记录在一块, PH的记录在一块.

二级索引

Sname	ID
Abby	3
Ed	4
Elsa	1
Nick	2

Student (聚簇索引)

ID	Sno	Sname	Ssex	Sage	Sdept
1	CS-001	Elsa	F	19	CS
2	PH-001	Nick	M	20	Physics
3	MA-001	Abby	F	18	Math
4	CS-002	Ed	M	19	CS

设计技巧6: 覆盖索引

Definition (覆盖索引)

如果一个索引包含(覆盖)一个查询需要用到的所有属性，则称该索引为覆盖索引(coverage index)

Example (覆盖索引)

- 查询: `SELECT Sno FROM Student WHERE Sname = 'Elsa';`
- 索引: `CREATE INDEX idx ON Student (Sname, Sno);`
- 索引idx能够覆盖上述查询

覆盖索引

Sname	Sno	地址	地址
Abby	MA-001	$addr_3$	$addr_1$
Ed	CS-002	$addr_2$	$addr_2$
Elsa	CS-001	$addr_1$	$addr_3$
Nick	PH-001	$addr_4$	$addr_4$

Student

Sno	Sname	Ssex	Sage	Sdept
CS-001	Elsa	F	19	CS
CS-002	Ed	M	19	CS
MA-001	Abby	F	18	Math
PH-001	Nick	M	20	Physics

设计技巧6: 覆盖索引(续)

- 索引项数量通常远小于元组数量，如果只需访问覆盖索引，则可以极大减少数据访问量
- 覆盖索引中属性值是顺序存储的，能更快找到满足条件的属性值
- 使用覆盖索引，则无需回表

Example (覆盖索引)

- 查询: `SELECT Sno FROM Student WHERE Sname = 'Elsa';`
- 索引: `CREATE INDEX idx ON Student (Sname, Sno);`
- 索引idx能够覆盖上述查询

覆盖索引

Sname	Sno	地址	地址
Abby	MA-001	$addr_3$	$addr_1$
Ed	CS-002	$addr_2$	$addr_2$
Elsa	CS-001	$addr_1$	$addr_3$
Nick	PH-001	$addr_4$	$addr_4$

Student

Sno	Sname	Ssex	Sage	Sdept
CS-001	Elsa	F	19	CS
CS-002	Ed	M	19	CS
MA-001	Abby	F	18	Math
PH-001	Nick	M	20	Physics

设计技巧6: 覆盖索引(续)

- 即使一个索引不能覆盖某个查询，我们也可以将该索引用作覆盖索引，并采用延迟连接(deferred join)方式改写查询

Example (延迟连接)

- 查询: `SELECT * FROM Student WHERE Sname = 'Elsa';`
- 索引: `CREATE INDEX idx ON Student (Sname, Sno);`
- 索引idx不能覆盖上述查询，但我们可以将该查询改写为

```
SELECT * FROM Student NATURAL JOIN
(SELECT Sno FROM Student WHERE Sname = 'Elsa') R;
```

- 可以利用覆盖索引快速执行子查询(延迟连接什么情况下有用?)

覆盖索引

Sname	Sno	地址	地址
Abby	MA-001	$addr_3$	$addr_1$
Ed	CS-002	$addr_2$	$addr_2$
Elsa	CS-001	$addr_1$	$addr_3$
Nick	PH-001	$addr_4$	$addr_4$

Student

Sno	Sname	Ssex	Sage	Sdept
CS-001	Elsa	F	19	CS
CS-002	Ed	M	19	CS
MA-001	Abby	F	18	Math
PH-001	Nick	M	20	Physics

避免不合理地使用索引

- 一方面要设计好的索引，另一方面要避免不合理地使用索引
- 若不合理地使用索引，则不但不会让查询变快，反而会使查询变慢

Example (类型转换)

- Student关系的Sno属性是字符串型
- Sno属性上建了主索引
- 下面的查询变慢了

```
SELECT * FROM Student WHERE Sno = 123456;
```

- 原因:
  1. DBMS不会使用主索引，而要进行全表扫描
  2. 因为Sno属性值为字符串，需要转换为整型

查询改写

- 查询优化器不能保证总是找到好的查询计划
- 用户不能给DBMS指定查询计划
- 用户只能通过添加索引或改写查询来影响查询优化器的决策
- 如果基于现有索引对查询进行改写就能获得好的查询计划，就没必要添加新的索引
- 如何改写查询取决于查询优化器的实现

Example (查询改写)

```
SELECT Sno, Sname FROM Student
WHERE Ssex = 'F' OR Sage = 19;
```

如果在Ssex和Sage上分别建有索引，则可以将该查询改写为

```
(SELECT Sno, Sname FROM Student WHERE Ssex = 'F') UNION
(SELECT Sno, Sname FROM Student WHERE Sage = 19);
```

6.3 物理存储结构的设计

Design of Physical Storage Structures

数据类型的选择

- 选择合理的数据类型对于提高数据库系统的性能非常重要

选择数据类型的原则

1. 尽量使用可以正确存储数据的最小数据类型
   - 原因: 最小数据类型占用空间更少，处理速度更快
   - 例: INTEGER或INT占4字节；SMALLINT占2字节；TINYINT占1字节；MEDIUMINT占3字节；BIGINT占8字节
   - 例: 使用VARCHAR(5)和VARCHAR(100)来存储'hello'的空间开销是一样的；但在排序时，MySQL会按照类型分配固定大小的内存块
2. 尽量选择简单的数据类型
   - 原因: 简单数据类型的处理速度更快
   - 例: 用DATE、DATETIME等类型来存储日期和时间，不要用字符串
   - 例: 用整型来存储IP地址，而不是用字符串
3. 若无需存储空值，则最好将属性声明为NOT NULL
   - 原因: 含空值的属性使得索引、统计、比较都更复杂

标识符类型的选择

- 为标识符属性选择合适的数据类型非常重要
  - 标识符属性通常会被当作索引属性，频繁地进行比较
  - 标识符属性通常会被当作主键或外键，频繁地进行连接
  - 在设计时，既要考虑标识符属性类型占用的空间，还要考虑比较的效率
- 整型：最好的选择 ✓
  - 占用空间少
  - 比较速度快
  - 可声明为AUTO_INCREMENT，为应用提供便利
- ENUM和SET类型：糟糕的选择
  - MySQL内部用整型来存储ENUM和SET类型的值，占用空间少
  - 在比较时会被转换为字符串，比较速度慢
- 字符串型：糟糕的选择
  - 占用空间大
  - 比较速度慢

关系模式的设计

- 一个关系不要包含太多属性，可以纵向拆分为多个关系

$$R(ID, A_1, A_2, \ldots, A_{1000})$$

- 不要使用“实体–属性–值(EAV)”设计模式(即为实体的每个属性建立一个关系)，因为在这种关系数据库模式上要进行大量连接

$$R_1(ID, A_1)$$
$$R_2(ID, A_2)$$
$$\ldots$$
$$R_{1000}(ID, A_{1000})$$

- 可以进行合理的S纵向拆分

$$R_1(ID, A_1, A_2, \ldots, A_{50})$$
$$R_2(ID, A_{51}, A_{52}, \ldots, A_{100})$$
$$\ldots$$
$$R_{20}(ID, A_{951}, A_{952}, \ldots, A_{1000})$$

关系模式的纵向划分(Vertical Partitioning)

将多属性分拆 Vertical Partitioning

- 根据工作负载对关系模式进行纵向划分

Example (关系模式的纵向划分)

关系模式Student(Sno, Sname, Ssex, Sage, Sdept, Balance)

- Balance表示学生校园账户的余额，频繁变化
- 经常按系查询学号和姓名

可以将Student划分为三个关系模式R1(Sno, Sname, Ssex, Sage), R2(Sno, Sdept), R3(Sno, ~~Sdept~~ Balance)

- 可以在更小的关系R2上更快地按系查询学号和姓名
- 在R3上更新Balance时，与R1和R2上的并发查询不产生冲突

关系模式的横向划分(Horizontal Partitioning)

- 根据工作负载，将关系中的元组划分到多个具有相同模式的关系中

Example (关系模式的横向划分)

关系模式Student(Sno, Sname, Ssex, Sage, Sdept)上查询的WHERE子句中经常含有Sdept = val的条件

- 将Student中的元组按Sdept属性值划分到不同关系中，如CS_Student, MA_Student, PH_Student
- 将含有条件Sdept = 'CS'的查询改写成CS_Student关系上的查询
- 创建视图Student

```
CREATE VIEW Student AS (
(SELECT * FROM CS_Student) UNION
(SELECT * FROM MA_Student) UNION
(SELECT * FROM PH_Student));
```

- 全体学生上的查询在视图Student上完成

总结

1. 物理数据库设计的步骤
2. 索引的设计
   - 索引的构成
   - 索引的分类
   - 索引数据结构
   - 索引设计技巧
   - 查询改写
3. 物理存储结构的设计
   - 数据类型的选择
   - 数据库的划分

练习 I

使用test_index.sql在MySQL上创建一个数据库

1. 使用show index命令查看该数据库上创建了哪些索引
2. 使用explain命令分析MySQL如何执行下列查询，验证索引的作用
   1. SELECT * FROM Foo WHERE b = 123 AND c = 23;
   2. SELECT * FROM FooIdx WHERE b = 123 AND c = 23;
   3. SELECT * FROM Foo WHERE b = 123;
   4. SELECT * FROM FooIdx WHERE b = 123;
   5. SELECT * FROM FooIdx WHERE c = 23;
   6. SELECT * FROM Foo WHERE tag LIKE '00123%';
   7. SELECT * FROM FooIdx WHERE tag LIKE '00123%';
   8. SELECT * FROM Foo WHERE b BETWEEN 123 AND 234;
   9. SELECT * FROM FooIdx WHERE b BETWEEN 123 AND 234;
   10. SELECT * FROM FooIdx
       WHERE b = 123 AND c BETWEEN 23 AND 45;
   11. SELECT * FROM FooIdx WHERE b = 123 AND c + 1 = 24;
   12. SELECT * FROM FooIdx WHERE a = 1234 AND c = 34;

练习 II

```
13 SELECT * FROM FooIdx
   WHERE a = 1234 AND b BETWEEN 234 AND 345 AND c = 34;
14 SELECT * FROM FooIdx WHERE tag LIKE '00123%';
15 SELECT * FROM FooIdx WHERE tag LIKE '0012%';
16 SELECT * FROM FooIdx WHERE tag LIKE '001234%' OR b = 56;
17 SELECT c FROM FooIdx WHERE b = 123;
18 SELECT a FROM FooIdx WHERE b = 123;
19 SELECT id FROM FooIdx WHERE b = 123;
```


参考资料

- Baron Schwartz, Peter Zaitsev, Vadim Tkachenko. High Performance MySQL.