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.edu.cn

2022年春

教学内容[1]

1. 物理数据库设计概述
2. 工作负载分析
3. 索引设计
4. 内模式设计
5. 查询改写
6. 概念模式调优

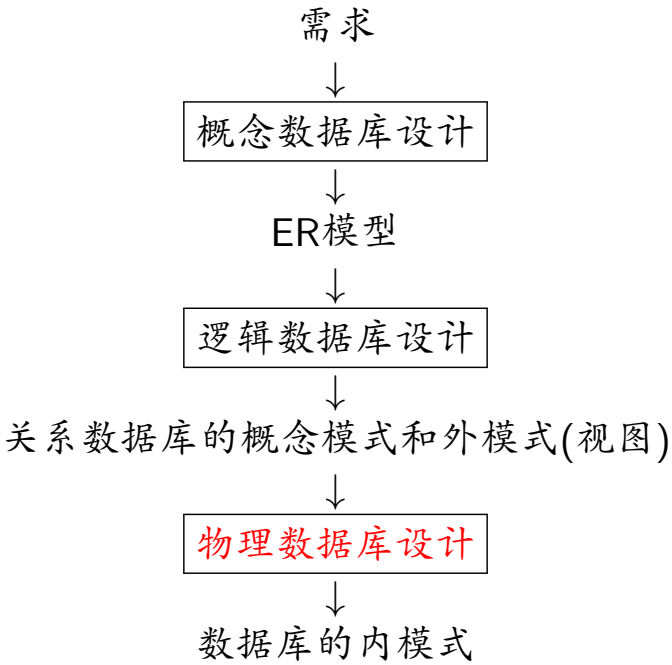
[1]课件更新于2022年4月2日

6.1 物理数据库设计概述

Overview of Physical Database Design

物理数据库设计(Physical Database Design)

在逻辑数据库设计的基础上，为数据库中的关系选择合适的存储结构和存取方法(access method)，并进行数据库调优(database tuning)，使数据库上的事务能够高效执行，满足用户对数据库系统的性能需求



物理数据库设计的任务

1. 工作负载分析: 分析工作负载中影响物理数据库设计的因素
2. 索引设计: 为关系选择合适的存取方法(主要是索引)
3. 内模式设计: 设计数据库的物理存储结构
4. 查询改写: 改写低效的查询
5. 概念模式调优: 从提高数据库性能(而非规范性)的角度，调整数据库的概念模式

6.2 工作负载分析

Workload Analysis

工作负载(Workload)

工作负载是一组混合在一起的查询和更新

查询列表

- 查询语句
- 查询出现的频率
- 用户对查询性能的要求

更新列表

- 更新语句
- 更新的出现频率
- 用户对更新性能的要求

查询分析

对于负载中每个查询，我们需要知道

- 查询涉及的关系
- 投影属性(SELECT子句中的属性)
- 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)

选择度(selectivity)

满足条件的结果元组在全部候选元组中所占的比例

更新分析

对于负载中每个更新，我们需要知道

- 被更新的关系
- 更新的类型(INSERT、DELETE、UPDATE)
- 被修改的关系上被修改的属性
- 更新条件涉及的关系
- 选择条件(selection condition)和连接条件(join condition)涉及的属性
- 选择条件和连接条件的选择度

6.3 索引设计

Index Design

索引设计的决策

1. 在哪个关系上建索引?
2. 索引键包含哪个(些)属性?
3. 索引支持什么类型的查询(等值查询、区间查询)?
4. 索引的类型(B+树、哈希表)
5. 索引是聚簇(clustered)索引还是非聚簇(nonclustered)索引?

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


索引设计的基本过程

依次检查工作负载中每个重要的查询Q(包括更新操作的查询部分)

1. 获取查询优化器为Q制定的查询执行计划
2. 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(第8章)、查询执行(第9章)和查询优化(第10章)
3. 如果可以，并且索引/的更新不会显著降低工作负载中更新操作的效率，则将索引/作为候选

索引设计的准则

索引设计准则1(是否建索引)

- 只需在查询涉及到的关系上建索引
- 如果工作负载无法获益，则不建索引
- 建一个索引最好能加速多个查询

索引设计的准则(续)

索引设计准则2(索引键的确定)

- 索引键只需在连接条件和选择条件涉及的属性中选择
- 等值选择条件 $\Longrightarrow$ 选择属性上的哈希表
- 区间选择条件 $\Longrightarrow$ 选择属性上的B+树

Example (索引键的确定)

1. ```
   SELECT * FROM Student WHERE Sname = 'Elsa';
   CREATE INDEX idx_sname ON Student(Sname) USING HASH;
   ```
2. ```
   SELECT * FROM Student WHERE Sage BETWEEN 18 AND 19;
   CREATE INDEX idx_sage ON Student(Sage) USING BTREE;
   ```

索引设计的准则(续)

索引设计准则3(复合索引的设计)

- WHERE子句中包含同一关系中多个属性上的选择条件 $\Longrightarrow$ 复合索引(composite index, 多个属性上的索引)
- 包含区间选择条件 $\Longrightarrow$ 复合索引键中属性的顺序
- 考虑设计覆盖索引(covering index)

Example (复合索引的设计)

1. ```
   SELECT * FROM Student
   WHERE Ssex = 'M' AND Sdept = 'CS';
   CREATE INDEX idx_ssex_sdept ON Student(Sdept, Ssex);
   ```
2. ```
   SELECT * FROM Student
   WHERE Sdept = 'CS' AND Sage BETWEEN 18 AND 19;
   CREATE INDEX idx_sdept_sage ON Student(Sdept, Sage);
   ```


B+树支持的查询 I

Example (B+树支持的查询)

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

Sname	Sage	Ssex	Address
Cindy	19	F	addr1
Ed	18	M	addr2
Elsa	19	F	addr3
Elsa	19	M	addr4
Elsa	20	F	addr5
Fawn	18	F	addr6

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;
   ```
3. 匹配属性前缀: 只匹配前缀属性的前缀部分

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

B+树支持的查询 II

4. 范围匹配: 在给定范围内对前缀属性进行匹配

```
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+树还支持按索引属性排序

```
SELECT * FROM Student ORDER BY Sname, Sage;
```

Example (B+树支持的查询)

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

Sname	Sage	Ssex	Address
Cindy	19	F	addr1
Ed	18	M	addr2
Elsa	19	F	addr3
Elsa	19	M	addr4
Elsa	20	F	addr5
Fawn	18	F	addr6

B+树的限制1

必须从B+树的最左属性开始匹配

Example (B+树的限制1)

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

不能在B+树上执行这个查询 ▸ 演示

Sname	Sage	Ssex	Address
Cindy	19	F	addr1
Ed	18	M	addr2
Elsa	19	F	addr3
Elsa	19	M	addr4
Elsa	20	F	addr5
Fawn	18	F	addr6

原因

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

B+树的限制2

条件中不能包含表达式

Example (B+树的限制2)

```
SELECT * FROM Student
WHERE Sname = 'Elsa' AND 2022 - Sage = 2003;
```

在B+树上只能根据条件Sname = 'Elsa'进行查找，然后在返回的元组上验证条件2020 - Sage = 2003 演示

Sname	Sage	Ssex	Address
Cindy	19	F	addr1
Ed	18	M	addr2
Elsa	19	F	addr3
Elsa	19	M	addr4
Elsa	20	F	addr5
Fawn	18	F	addr6

B+树的限制3

不能跳过B+树中的属性

Example (B+树的限制3)

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

在B+树上只能根据条件Sname = 'Elsa'进行查找，然后在返回的元组上验证条件Ssex = 'F' 演示

Sname	Sage	Ssex	Address
Cindy	19	F	addr1
Ed	18	M	addr2
Elsa	19	F	addr3
Elsa	19	M	addr4
Elsa	20	F	addr5
Fawn	18	F	addr6

B+树的限制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(该限制条件并非在所有DBMS上都成立，如MySQL)

Example (B+树的限制4)

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

在B+树上只能根据条件Sname = 'Elsa' AND Sage BETWEEN 18 AND 20进行查找，然后在返回的元组上验证条件Ssex = 'F' 演示

Sname	Sage	Ssex	Address
Cindy	19	F	addr1
Ed	18	M	addr2
Elsa	19	F	addr3
Elsa	19	M	addr4
Elsa	20	F	addr5
Fawn	18	F	addr6

覆盖索引(Covering Index)

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

- 只需使用覆盖索引即可，无需回表
- 索引项大小通常远小于元组大小，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

Example (覆盖索引)

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

覆盖索引

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

地址

$addr_1$
$addr_2$
$addr_3$
$addr_4$

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

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)的方式改写查询

Example (延迟连接)

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

```
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
Abby	MA-001	$addr_3$
Ed	CS-002	$addr_2$
Elsa	CS-001	$addr_1$
Nick	PH-001	$addr_4$

地址

$addr_1$
$addr_2$
$addr_3$
$addr_4$

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

复合索引中属性的顺序

在属性Sname、Sage和Ssex上可以建立6种不同的索引，哪种好?

1. `CREATE INDEX idx1 ON Student (Sname, Sage, Ssex);`
2. `CREATE INDEX idx2 ON Student (Sname, Ssex, Sage);`
3. `CREATE INDEX idx3 ON Student (Sage, Sname, Ssex);`
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，将选择度最高的属性放在最前面通常是好的，这样可以更快过滤掉不满足条件的元组

单个复合索引 vs. 多个单属性索引

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

方案1: 建立一个复合索引

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

- 该索引能直接支持上面的查询

单个复合索引 vs. 多个单属性索引(续)

方案2: 建立多个单属性索引

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

1. 在idx1上查找满足Sname = 'Elsa'的元组地址
2. 在idx2上查找满足Sage = 19的元组地址
3. 在idx3上查找满足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

多个单属性索引的缺点

- 没有在单个复合索引上做查询效率高
- 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

索引设计的准则(续)

索引设计准则4(是否使用聚簇索引)

- 一个关系上至多能有一个聚簇索引(clustered index)
- 区间查询 $\Longrightarrow$ 聚簇索引(聚簇索引对区间查询的支持最好)
- 覆盖索引(covering index)无需是聚簇索引，因为无需回表

Example (聚簇索引)

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

聚簇索引

Sname	Sage	Ssex	Address
Cindy	19	F	addr1
Ed	18	M	addr2
Elsa	19	F	addr3
Elsa	19	M	addr4
Elsa	20	F	addr5
Fawn	18	F	addr6

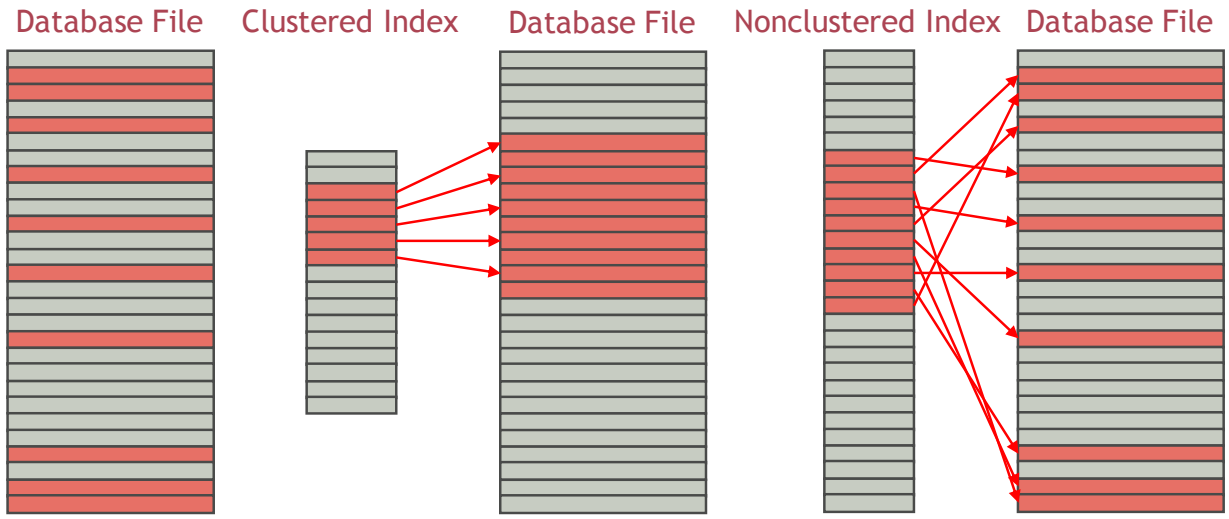
Address
addr1
addr2
addr3
addr4
addr5
addr6

Student

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

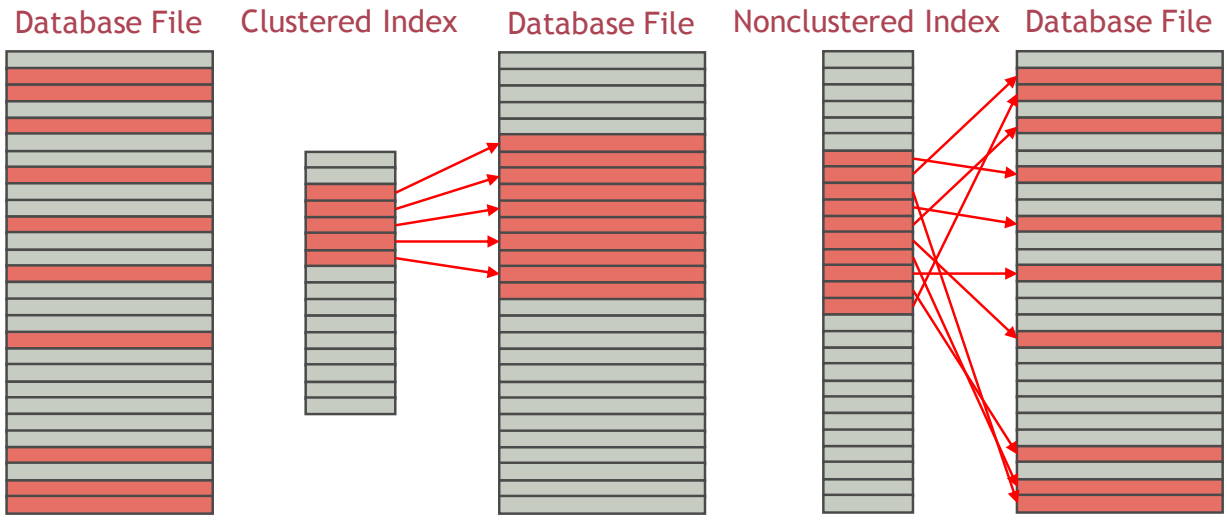
不同存取路径的区别

- 顺序扫描: $B(R)$次I/O
- 聚簇索引: $\theta \cdot B(R)$次I/O，其中θ是查询条件的选择度
- 非聚簇索引: $\theta \cdot T(R)$次I/O，其中θ是查询条件的选择度



不同存取路径的区别(续)

- 当条件选择度很高(θ很小)时，聚簇索引$\prec$非聚簇索引$\prec$顺序扫描
- 当条件选择度低(θ较大)时，聚簇索引$\prec$顺序扫描$\prec$非聚簇索引



索引设计的准则(续)

索引设计准则5(用哈希表还是B+树)

- B+树通常都是更好的选择(支持等值查询和范围查询)
- 如果在一个关系R上建索引的目的只是为了加快R和S的嵌套循环连接(nested loop join)，且R是内关系(inner relation)，则可以在R的连接属性上建哈希表
- 如果有一个特别重要的等值查询，且在搜索键上没有范围查询，则可以建哈希表

Example (用哈希表还是B+树)

```
CREATE INDEX idx_sno ON Student(Sno) USING HASH;
```

1. `SELECT * FROM Student NATURAL JOIN SC;`
2. `SELECT * FROM Student WHERE Sno = 'CS-001';`

哈希索引的限制

```
CREATE INDEX idx ON Student (Sname, Sage, Ssex) 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 Studen 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

  SELECT * FROM Student
  WHERE Sname = 'Elsa' AND Sage = 19 AND Ssex = 'F'
  ORDER BY Sno; (不能使用索引)
- 哈希索引存在冲突问题

索引设计的准则(续)

索引设计准则6(权衡索引维护代价)

- 如果维护一个索引导致工作负载中很多频繁执行的更新操作变慢，则删除该索引

索引设计准则7(不唯准则)

- 要根据数据分布的特点、工作负载的特点、数据库上已有的索引，灵活地设计新索引

索引调优(Index Tuning)

重建索引

- 索引可能因系统故障而损坏
- 索引在反复更新后可能出现空间碎片，导致索引变得"臃肿"
- 修改了索引的参数(如填充因子)，并希望这种修改完全生效
- 使用REINDEX命令重建索引(PostgreSQL和openGauss)

定期更新数据库的统计信息

- 查询优化器根据代价估计模型估计的代价决定是否使用索引
- 过时的统计信息会降低代价估计模型的准确性
- 使用ANALYZE命令收集数据库的统计信息

MySQL索引设计技巧1: 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));
```

Demo

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

MySQL索引设计技巧1: 前缀索引(续)

Example (前缀索引，前缀长度= 2，选择性= 1 ⋆)

```
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

MySQL索引设计技巧1: 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)

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？

MySQL索引设计技巧2: 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

MySQL索引设计技巧2: 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

MySQL索引设计技巧2: 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)

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

MySQL索引设计技巧3: 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

MySQL索引设计技巧3: 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

MySQL的EXPLAIN命令

- 用法: `EXPLAIN` SQL语句
- 功能: 解释MySQL如何执行该语句
- 输出结果:
  - type: 执行SQL语句时使用的数据访问方式。ALL (全表扫描) < index (索引扫描) < range (索引上的范围扫描) < ref (索引访问，返回满足条件的多个元组) < eq_ref (索引访问，最多只返回满足条件的一条元组) < const (最多只返回满足条件的一条元组，查询前访问元组，并将其转为常量)
  - possible_keys: 执行SQL语句时可能用到的索引
  - key: 执行SQL语句时真正使用的索引
  - key_len: 访问索引时使用的键的长度
  - rows: 预计访问元组数
  - Extra: 查询执行过程的一些详细信息。Using where (使用WHERE从句来过滤元组)、Using index condition (索引条件推送)、Using index (使用覆盖索引)

基于人工智能的索引设计

使用人工智能技术，根据数据分布和工作负载的特点，优化选择索引

Database Meets AI: A Survey

Xuanhe Zhou, Chengliang Chai, Guoliang Li, Ji Sun

Abstract—Database and Artificial Intelligence (AI) can benefit from each other. On one hand, AI can make database more intelligent (AI4DB). For example, traditional empirical database optimization techniques (e.g., cost estimation, join order selection, knob tuning, index and view selection) cannot meet the high-performance requirement for large-scale database instances, various applications and diversified users, especially on the cloud. Fortunately, learning-based techniques can alleviate this problem. On the other hand, database techniques can optimize AI models (DB4AI). For example, AI is hard to deploy in real applications, because it requires developers to write complex codes and train complicated models. Database techniques can be used to reduce the complexity of using AI models, accelerate AI algorithms and provide AI capability inside databases. Thus both DB4AI and AI4DB have been extensively studied recently. In this paper, we review existing studies on AI4DB and DB4AI. For AI4DB, we review the techniques on learning-based configuration tuning, optimizer, index/view advisor, and security. For DB4AI, we review AI-oriented declarative language, AI-oriented data governance, training acceleration, and inference acceleration. Finally, we provide research challenges and future directions.

Index Terms—Database, Artificial Intelligence, DB4AI, AI4DB

1 Introduction

Artificial intelligence (AI) and database (DB) have been extensively studied over the last five decades. First, database systems have been widely used in many applications, because databases are easy to use by providing user-friendly declarative query paradigms and encapsulating complicated query optimization functions. Second, AI has recently made breakthroughs due to three driving forces: large-scale data, new algorithms and high computing power. Moreover, AI and database can benefit from each other [144], [55]. On one hand, AI can make database more intelligent (AI4DB). For example, traditional empirical database optimization techniques (e.g., cost estimation, join order selection, knob tuning, index and view selection) cannot meet the high-performance requirement for large-scale database instances, various applications and diversified users, especially on the cloud. Fortunately, learning-based techniques can alleviate this problem. For instance, deep learning can improve the quality of cost estimation and deep reinforcement learning can be used to tune database knobs. On the other hand, database techniques can optimize AI models (DB4AI). AI is hard to deploy in real applications, because it requires developers to write complex codes and train complicated models. Database techniques can be used to reduce the complexity of using AI models, accelerate AI algorithms and provide AI capability inside databases. Thus both DB4AI and AI4DB have been extensively studied recently.

1.1 AI for DB

Traditional database design is based on empirical methodologies and specifications, and requires human involvement (e.g., DBAs) to tune and maintain the databases [76], [17]. AI techniques are used to alleviate these limitations—exploring more design space than humans and replacing heuristics to address hard problems. We categorize existing techniques of using AI to optimize DB as below.

• *Xuanhe Zhou, Chengliang Chai, Guoliang Li, Ji Sun were with the Department of Computer Science, Tsinghua University, Beijing, China. Corresponding author: Guoliang Li, Chengliang Chai.*

Learning-based Database Configuration. (1) Knob tuning. Databases have hundreds of knobs and it requires DBAs to tune the knobs so as to adapt to different scenarios. Obviously, DBAs are not scalable to millions of database instances on cloud databases. Recently the database community attempts to utilize learning-based techniques [3], [77], [157] to automatically tune the knobs, which can explore more knob combination space and recommend high-quality knob values, thus achieving better results than DBAs. (2) Index/View advisor. Database indexes and views are fairly crucial to achieve high performance. However, traditional databases rely highly on DBAs to build and maintain indexes and views. As there are a huge number of column/table combinations, it is expensive to recommend and build appropriate indexes/views. Recently, there are some learning-based works that automatically recommend and maintain the indexes and views. (3) SQL Rewriter. Many SQL programmers cannot write high-quality SQLs and it requires to rewrite the SQL queries to improve the performance. For example, the nested queries will be rewritten into join queries to enable SQL optimization. Existing methods employ rule-based strategies, which uses some predefined rules to rewrite SQL queries. However, these rule-based methods rely on high-quality rules and cannot be scale to a large number of rules. Thus, deep reinforcing learning can be used to judiciously select the appropriate rules and apply the rules in a good order.

Learning-based Database Optimization. (1) Cardinality/Cost Estimation. Database optimizer relies on cost and cardinality estimation to select an optimized plan, but traditional techniques cannot effectively capture the correlations between different columns/tables and thus cannot provide high-quality estimation. Recently deep learning based techniques [64], [110] are proposed to estimate the cost and cardinality which can achieve better results, by using deep neural networks to capture the correlations. (2) Join order selection. A SQL query may have millions, even billions of possible plans and it is very important to efficiently

X. Zhou, C. Chai, G. Li, J. Sun. **Database Meets Artificial Intelligence: A Survey**. *IEEE Transactions on Knowledge and Data Engineering*, 34(3):1096–1116, 2022.

Section 2.1.2 “Index Selection”

6.4 内模式设计

Designing the Internal Schema

内模式设计的决策

1. 关系是否采用聚簇存储?
2. 如果使用聚簇存储，应按哪些属性聚簇存储?

内模式设计的准则

该设计准则同聚簇索引的设计准则

- 聚簇索引决定了关系按哪些属性聚簇存储

Example (聚簇关系 ▸ 演示)

```
CREATE INDEX idx_sname ON Student (Sname);
CLUSTER Student USING idx_sname; -- PostgreSQL和openGauss
```

6.5 查询改写
Rewriting Queries

查询改写(Query Rewrite)

如果通过改写查询就能获得好的查询计划，则没必要增加新的索引

- 查询优化器不能保证总是找到好的查询计划
- 用户基本上不能给DBMS指定查询计划
- 用户可以通过添加索引或改写查询来影响查询优化器的决策

查询改写的案例1

Example (恒真/假的选择条件)

改写前:

1. `SELECT * FROM Student WHERE 1 = 1;`
2. `SELECT * FROM Student WHERE 1 = 0;`

改写原因:

- 选择条件恒为真/假
- 大量CPU资源浪费在过滤元组上

改写后:

1. `SELECT * FROM Student;`
2. 无结果

查询改写的案例2

Example (含表达式的选择条件)

改写前:

```
SELECT * FROM Student WHERE 2022 - Sage = 2003;
```

改写原因:

- 选择条件含有表达式
- 查询优化器无法确定使用哪个索引

改写后:

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

查询改写的案例3

Example (含OR的选择条件)

改写前:

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

改写原因:

- 没有索引可以支持该选择条件
- 如果在Ssex和Sage上分别建有索引，则改写查询可能使DBMS使用索引合并(index merge)

改写后:

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


查询改写的案例4

Example (无用的去重操作)

改写前:

```
SELECT DISTINCT Sno, Sname FROM Student WHERE Ssex = 'F';
```

改写原因:

- 元组去重操作很昂贵
- 因为SELECT子句中包含候选键Sno，所以DISTINCT画蛇添足

改写后:

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

查询改写的案例5

Example (无用的分组操作)

改写前:

```
SELECT MIN(Sage) FROM Student
GROUP BY Sdept HAVING Sdept = 'CS';
```

改写原因:

- 分组操作很昂贵
- 查询中的分组操作毫无必要

改写后:

```
SELECT MIN(Sage) FROM Student WHERE Sdept = 'CS';
```


查询改写的案例6

Example (无用的投影操作)

改写前:

```
SELECT * FROM SC WHERE EXISTS (
  SELECT Sno FROM Student
  WHERE Sname = 'Elsa' AND Student.Sno = SC.Sno);
```

改写原因:

- EXISTS只检查子查询结果是否为空
- 子查询的投影操作没有必要

改写后:

```
SELECT * FROM SC WHERE EXISTS (
  SELECT * FROM Student
  WHERE Sname = 'Elsa' AND Student.Sno = SC.Sno);
```

查询改写的案例7

Example (无用的连接操作)

改写前:

```
SELECT Sno, Cno, Grade FROM Student NATURAL JOIN SC;
```

改写原因:

- SC中每条元组一定能够与Student中的元组连接
- SELECT子句中的属性全部来自SC，故连接操作没有必要

改写后:

```
SELECT * FROM SC;
```


查询改写的案例8

Example (临时关系)

改写前:

```
SELECT * INTO Temp FROM Student NATURAL JOIN SC
WHERE Sdept = 'CS';

SELECT Cno, MAX(Grade) FROM Temp GROUP BY Cno;
```

改写原因:

- 物化(materialize)临时关系Temp占用额外的存储空间
- 临时关系Temp上没有索引，可能会浪费掉潜在的查询优化机会(假设已有SC(Cno, Grade)上的复合索引)

改写后:

```
SELECT Cno, MAX(Grade) FROM Student NATURAL JOIN SC
WHERE Sdept = 'CS' GROUP BY Cno;
```

查询改写的案例9

Example (嵌套查询)

改写前:

```
SELECT Cno FROM SC WHERE Sno IN (
  SELECT Sno FROM Student WHERE Sdept = 'CS');
```

改写原因:

- 查询优化器优化嵌套查询的能力通常较差
- 将嵌套查询改写为非嵌套查询通常会带来更优的查询执行计划

改写后:

```
SELECT Cno FROM Student NATURAL JOIN SC WHERE Sdept = 'CS';
```


查询改写的案例10

Example (类型转换)

改写前:

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

改写原因:

- Student关系的Sno属性是字符串型
- DBMS不会使用Sno上的索引，而要进行顺序扫描
- 因为Sno属性值为字符串，需要转换为整型

改写后:

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

基于人工智能的查询改写

使用人工智能技术，优化运用改写规则，提高查询改写质量

Database Meets AI: A Survey

Xuanhe Zhou, Chengliang Chai, Guoliang Li, Ji Sun

Abstract—Database and Artificial Intelligence (AI) can benefit from each other. On one hand, AI can make database more intelligent (AI4DB). For example, traditional empirical database optimization techniques (e.g., cost estimation, join order selection, knob tuning, index and view selection) cannot meet the high-performance requirement for large-scale database instances, various applications and diversified users, especially on the cloud. Fortunately, learning-based techniques can alleviate this problem. On the other hand, database techniques can optimize AI models (DB4AI). For example, AI is hard to deploy in real applications, because it requires developers to write complex codes and train complicated models. Database techniques can be used to reduce the complexity of using AI models, accelerate AI algorithms and provide AI capability inside databases. Thus both DB4AI and AI4DB have been extensively studied recently. In this paper, we review existing studies on AI4DB and DB4AI. For AI4DB, we review the techniques on learning-based configuration tuning, optimizer, index/view advisor, and security. For DB4AI, we review AI-oriented declarative language, AI-oriented data governance, training acceleration, and inference acceleration. Finally, we provide research challenges and future directions.

Index Terms—Database, Artificial Intelligence, DB4AI, AI4DB

1 Introduction

Artificial intelligence (AI) and database (DB) have been extensively studied over the last five decades. First, database systems have been widely used in many applications, because databases are easy to use by providing user-friendly declarative query paradigms and encapsulating complicated query optimization functions. Second, AI has recently made breakthroughs due to three driving forces: large-scale data, new algorithms and high computing power. Moreover, AI and database can benefit from each other [144], [55]. On one hand, AI can make database more intelligent (AI4DB). For example, traditional empirical database optimization techniques (e.g., cost estimation, join order selection, knob tuning, index and view selection) cannot meet the high-performance requirement for large-scale database instances, various applications and diversified users, especially on the cloud. Fortunately, learning-based techniques can alleviate this problem. For instance, deep learning can improve the quality of cost estimation and deep reinforcement learning can be used to tune database knobs. On the other hand, database techniques can optimize AI models (DB4AI). AI is hard to deploy in real applications, because it requires developers to write complex codes and train complicated models. Database techniques can be used to reduce the complexity of using AI models, accelerate AI algorithms and provide AI capability inside databases. Thus both DB4AI and AI4DB have been extensively studied recently.

1.1 AI for DB

Traditional database design is based on empirical methodologies and specifications, and requires human involvement (e.g., DBAs) to tune and maintain the databases [76], [17]. AI techniques are used to alleviate these limitations—exploiting more design space than humans and replacing heuristics to address hard problems. We categorize existing techniques of using AI to optimize DB as below.

• *Xuanhe Zhou, Chengliang Chai, Guoliang Li, Ji Sun were with the Department of Computer Science, Tsinghua University, Beijing, China. Corresponding author: Guoliang Li, Chengliang Chai.*

Learning-based Database Configuration. (1) Knob tuning. Databases have hundreds of knobs and it requires DBAs to tune the knobs so as to adapt to different scenarios. Obviously, DBAs are not scalable to millions of database instances on cloud databases. Recently the database community attempts to utilize learning-based techniques [3], [77], [157] to automatically tune the knobs, which can explore more knob combination space and recommend high-quality knob values, thus achieving better results than DBAs. (2) Index/View advisor. Database indexes and views are fairly crucial to achieve high performance. However, traditional databases rely highly on DBAs to build and maintain indexes and views. As there are a huge number of column/table combinations, it is expensive to recommend and build appropriate indexes/views. Recently, there are some learning-based works that automatically recommend and maintain the indexes and views. (3) SQL Rewriter. Many SQL programmers cannot write high-quality SQLs and it requires to rewrite the SQL queries to improve the performance. For example, the nested queries will be rewritten into join queries to enable SQL optimization. Existing methods employ rule-based strategies, which uses some predefined rules to rewrite SQL queries. However, these rule-based methods rely on high-quality rules and cannot be scale to a large number of rules. Thus, deep reinforcing learning can be used to judiciously select the appropriate rules and apply the rules in a good order.

Learning-based Database Optimization. (1) Cardinality/-Cost Estimation. Database optimizer relies on cost and cardinality estimation to select an optimized plan, but traditional techniques cannot effectively capture the correlations between different columns/tables and thus cannot provide high-quality estimation. Recently deep learning based techniques [64], [110] are proposed to estimate the cost and cardinality which can achieve better results, by using deep neural networks to capture the correlations. (2) Join order selection. A SQL query may have millions, even billions of possible plans and it is very important to efficiently

X. Zhou, C. Chai, G. Li, J. Sun. **Database Meets Artificial Intelligence: A Survey**. *IEEE Transactions on Knowledge and Data Engineering*, 34(3):1096–1116, 2022.

Section 2.1.4 “SQL Rewrite”

6.6 概念模式调优

Tuning the Conceptual Schema

概念模式调优的决策

1. 使用高范式级别的设计方案还是低范式级别的设计方案?
2. 是否需要“反规范化(denormalize)”关系模式?
3. 是否需要继续分解一个BCNF关系模式?
4. 是否对关系模式进行水平分解(horizontal decomposition)?
5. 属性类型的调优

弱范式(Weaker Normal Form)模式

出于性能考虑，可以保留一个低范式关系模式，而不是将其分解为高范式关系模式

Example (弱范式)

Student2

Sno	Sname	Ssex	Sage	Sdept	Sdorm

- Sdorm: 学生居住的公寓
- 假设同一个系男同学住在同一个公寓，同一个系女同学住在同一个公寓，即$(Sdept, Ssex) \longrightarrow Sdorm$
- $Student2 \in 2NF$
- 出于规范化的考虑，应将Student2分解Student(Sno, Sname, Ssex, Sage, Sdept)和SD(Sdept, Ssex, Sdorm)
- 出于性能考虑，如果系统频繁查询学生居住的公寓，则可以保留Student2

反规范化(Denormalization)

为了提高查询性能，故意向高范式关系模式中添加依赖属性，降低范式

Example (反规范化)

Student

Sno	Sname	Ssex	Sage	Sdept

- $Student \in BCNF$
- Sdorm: 学生居住的公寓
- 假设同一个系男同学住在同一个公寓，同一个系女同学住在同一个公寓，即$(Sdept, Ssex) \longrightarrow Sdorm$
- 如果系统频繁查询学生居住的公寓，则可以将属性Sdorm加入Student中

Student2

Sno	Sname	Ssex	Sage	Sdept	Sdorm

竖直分解(Vertical Decomposition)

为了提高查询性能，可以继续分解一个BCNF关系模式

Example (竖直分解)

Student

Sno	Sname	Ssex	Sage	Sdept	Sbal

- $Student \in BCNF$
- Sbal: 学生校园卡账户余额，频繁变化
- 应用经常按系查询学生的学号和姓名

StuNSA

Sno	Sname	Ssex	Sage

StuND

Sno	Sname	Sdept

StuB

Sno	Sbal

- 在更小的关系上通常能更快地执行查询
- 在StuB上更新Sbal时，与StuNSA和StuND上的并发查询不冲突

水平分解(Horizontal Decomposition)

为了提高查询性能，将一个关系中的元组划分到多个不同的关系中

Example (水平分解)

- 将Sdept = 'CS'的元组存入新关系CSStudent
- 将Sdept = 'Math'的元组存入新关系MathStudent
- 将Sdept = 'Physics'的元组存入新关系PhysicsStudent
- 为了对应用程序屏蔽概念模式的变化，创建视图Student

```
CREATE VIEW Student(Sno, Sname, Ssex, Sage, Sdept) AS
  ((SELECT * FROM CSStudent) UNION
   (SELECT * FROM MathStudent) UNION
   (SELECT * FROM PhysicsStudent));
```

- SELECT * FROM Student WHERE Sdept = 'CS'并不会被DBMS自动翻译为SELECT * FROM CSStudent
- 为了效率，需要把水平分解的结果暴露给用户

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

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

总结

练习 I

使用index-demo.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;`
3. 通过分析EXPLAIN结果中key和key_len的值，说明以下查询是按照什么属性在索引上进行查找的:
   - `SELECT * FROM FooIdx WHERE id = 1234;`
   - `SELECT * FROM FooIdx WHERE a = 1234;`
   - `SELECT * FROM FooIdx WHERE b = 123 AND c + 1 = 24;`
   - `SELECT * FROM FooIdx WHERE a = 1234 AND c = 34;`
   - `SELECT * FROM FooIdx WHERE a = 1234 AND b BETWEEN 234 AND 345 AND c = 34;`

在PostgreSQL上重复上述实验，你有什么发现?

参考资料

6.2节(“索引设计”)中部分内容取自

- Baron Schwartz, Peter Zaitsev, Vadim Tkachenko. High Performance MySQL, 3rd Edition. O'Reilly Media, Inc, March 2012.