

Introduction to Data Management

Lecture 19 (Storage and Indexing)



Instructor: Mike Carey
mjcarey@ics.uci.edu

Database Management Systems 3ed, R. Ramakrishnan and J. Gehrke

1



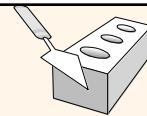
Announcements

- ❖ Midterm #2 next **Wednesday (5/22)** in class
 - Relational languages (see syllabus on wiki!)
 - Sample exam from last year is available online
- ❖ HW #6 due next **Monday at 7 PM**
 - One late “day” (**actually 22 hours**) will be an option
 - Solution will be posted on Tuesday after **5 PM**
- ❖ Today’s lecture plan
 - Storage, indexing, & beyond (last 1/3 of course!)
 - Not on Midterm #2, of course

Database Management Systems 3ed, R. Ramakrishnan and J. Gehrke

2

Midterm #2 Scope



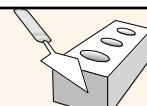
Syllabus

Topic	Reading
Databases and DB Systems	Ch. 1
Entity-Relationship (E-R) Data Model	Ch. 2.1-2.5, 2.8
Relational Data Model	Ch. 3.1-3.2
E-R to Relational Translation	3.5
Relational Design Theory	Ch. 19.1-19.6, 20.8
<i>Midterm Exam 1</i>	<i>Mon, Apr 29 (in class)</i>
Relational Algebra	Ch. 4.1-4.2
Relational Calculus	Ch. 4.3-4.4
SQL Basics (SPJ and Nested Queries)	Ch. 3.4, 5.1-5.3
SQL Analytics (Aggregation, Nulls, and Outer Joins)	Ch. 5.4-5.6
Advanced SQL Goodies (Constraints, Triggers, Views, and Security)	Ch. 3.3, 3.6, 5.7-5.9, 21.1-21.3, 21.7
<i>Midterm Exam 2</i>	<i>Wed, May 22 (in class)</i>
Tree-Based Indexing	Ch. 9.1, 8.1-8.3, 10.1-10.2
Hash-Based Indexing	Ch. 10.3-10.8, 11.1
Physical DB Design	Ch. 8.5, 20.1-20.7
Semistructured Data Management (<i>a.k.a.</i> NoSQL)	AsterixDB SQL++ Primer , Couchbase SQL++ Book
Basics of Transactions	Ch. 16 and Lecture Notes
<i>Endterm Exam</i>	<i>Fri, Jun 7 (in class)</i>

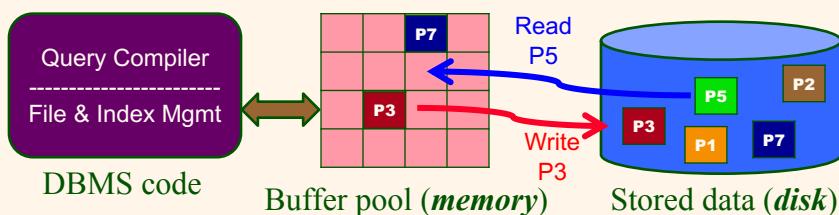
Database Management Systems 3ed, R. Ramakrishnan and J. Gehrke

3

Disks and Files (Reminder)



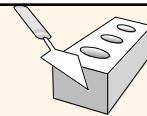
- ❖ DBMSs store data on *secondary storage*.
- ❖ This has major implications for DBMS design!
 - **READ:** xfer data from *disk* (or flash) to *memory* (RAM).
 - **WRITE:** xfer data from RAM to disk (or flash).
 - Both are high-cost operations, relative to in-memory operations, so must be considered carefully!



Database Management Systems 3ed, R. Ramakrishnan and J. Gehrke

4

Ex: Emp(eid, ename, sal, deptid)



P1 1 111 Smith 3K 1
 2 222 Lee 100K 3
 3 333 Carey 80K 1
 4 444 Smith 12K 7

P2 1 555 Smith 18K 3
 2 666 Jones 90K 5
 3 777 Smith 23K 4
 4 888 Krishan 60K 8

Underlying *Emp* file pages

P10000 1
 2
 3
 4 999999 Smith 18K 11

← Record id (RID) is (P2,3)

Database Management Systems 3ed, R. Ramakrishnan and J. Gehrke

5

Processing a Query

- ❖ Suppose someone asks a simple SQL query:
 - `SELECT * FROM Emp WHERE eid = 12345;`
- ❖ Some processing options would include:
 - *Option 1:* Sequentially scan the data file (and stop, if we know eid is a key) → 5000 page reads (avg.)
 - *Option 2:* Binary search the data file (and stop, if we know eid is a key) → $\log_2(10,000) \approx 15$ page reads (avg.)
 - Even though Option 2 is ≈ 30x faster, we'd like to do even better (especially for large data sets!)

Database Management Systems 3ed, R. Ramakrishnan and J. Gehrke

6

Indexing is the Answer!



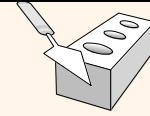
❖ Index maps from keys to associated info I

- $I(k)$ can be the *data record* with key k , or
- $I(k)$ can be the *RID* of the data record with key k , or
- $I(k)$ can be a *list of RIDs* of data records with key k !
- Alternatively, we could map from data field values to the *PK value(s)* of the associated record(s) - SQL Server and AsterixDB do this, for example.

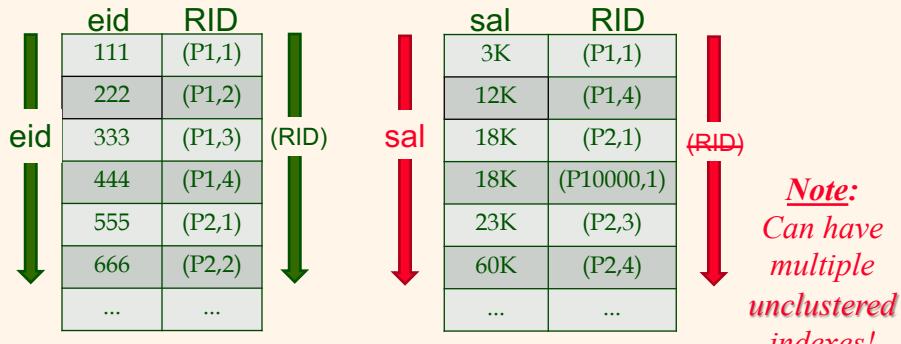
Indexes

- ❖ An *index* on a file speeds up selections on the *search key fields* for the index.
 - Any subset of the fields of a relation can serve as the search key for an index on the relation.
 - *Search key* is **not** the same as a "*key*" (i.e., it's not the primary key, it's just a field we're very interested in).
- ❖ An index contains a collection of *data entries*, and it supports efficient retrieval of *all* data entries k^* with a given key value k .
 - Given a data entry k^* , we can find 1st actual record with key k with just more disk I/O. (Details soon ...)

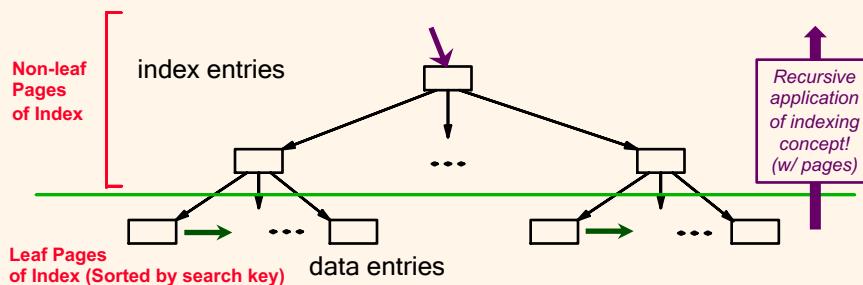
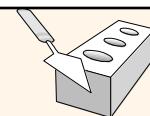
Ex: Emp (eid, ename, sal, deptid)



- ❖ One simple approach would be to have another file, sorted *on k*, for each *k* that we want to index
 - Hundreds of (key, RID) entries will fit on a *single page*
 - Index is thus *much* smaller than the data file
 - Less data (*fewer reads!*) to search to locate the RIDs of interest

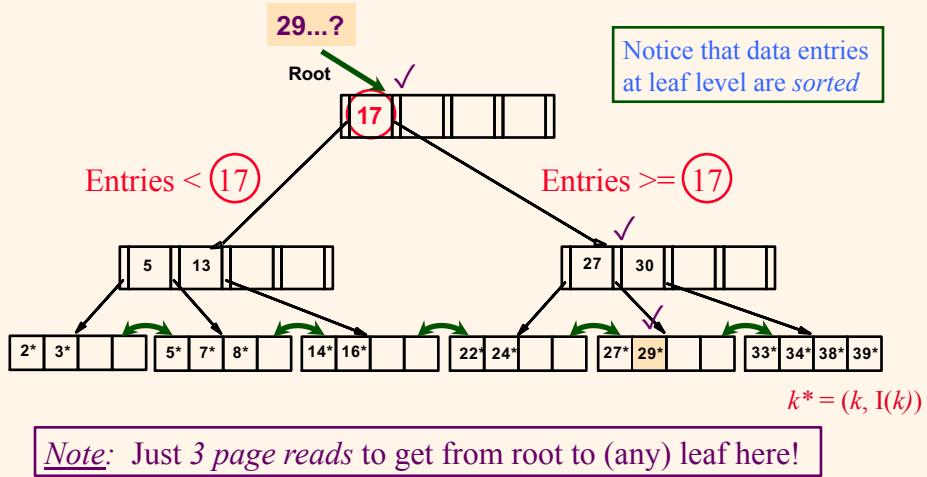


Even Better: Tree Indexes!



- ❖ Leaf pages contain *data entries*, and are chained
- ❖ Non-leaf pages have *index entries*; role is to guide searches
- ❖ Query processing steps become:
 1. Choose a good index to use (if one is available)
 2. Search the index to determine the interesting RID(s)
 3. Use the RID(s) to fetch the corresponding record(s)

An Example ($B+$ Tree)



Database Management Systems 3ed, R. Ramakrishnan and J. Gehrke

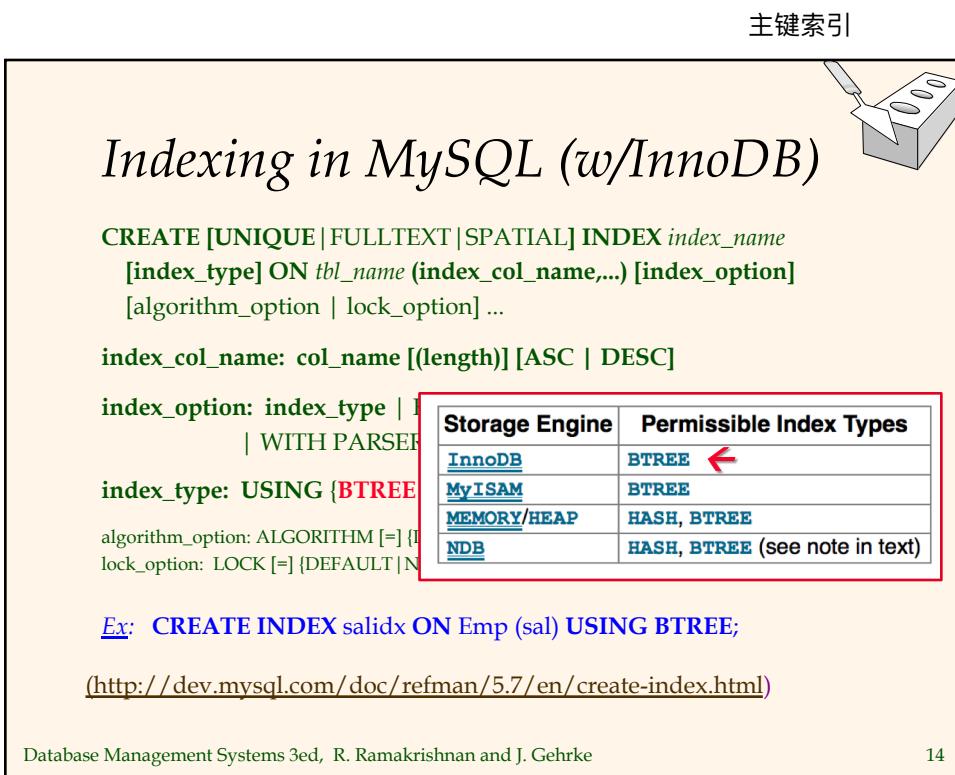
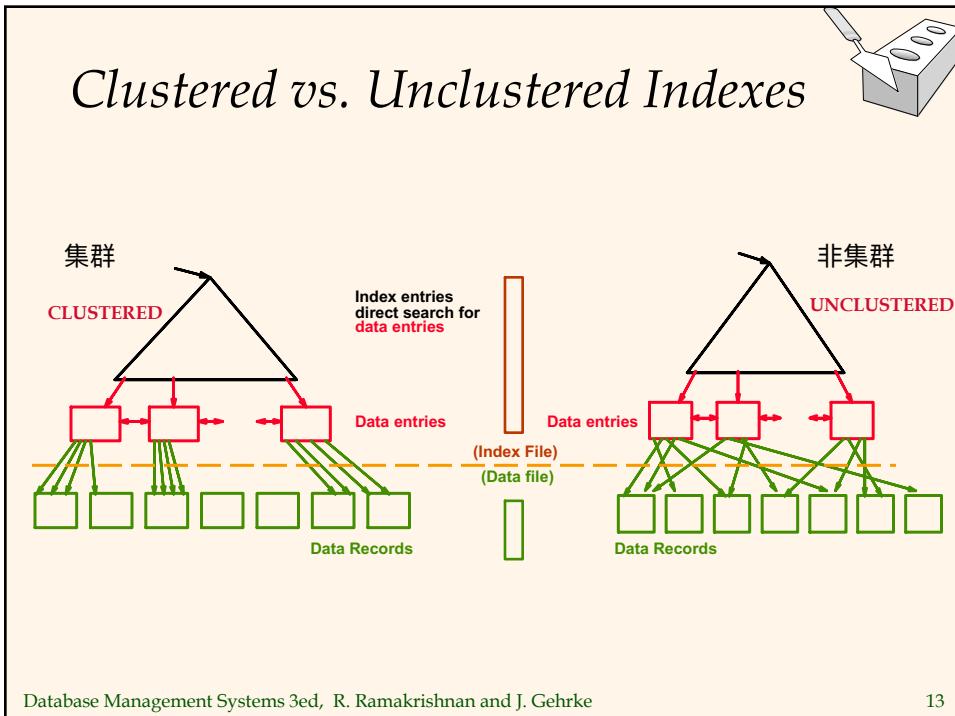
11

Index Classification

- ❖ **Primary vs. secondary:** If search key contains the primary key, it's called the primary index.
 - *Unique* index: Search key contains a *candidate* key.
- ❖ **Clustered vs. unclustered:** If order of data records is the same as, or “close to”, the order of stored data records, then it's called a clustered index.
 - A table can be clustered on *at most one* search key (see RID ordering in example a few slides ago).
 - Cost of retrieving data records via an index varies *greatly* based on whether index is clustered or not!

Database Management Systems 3ed, R. Ramakrishnan and J. Gehrke

12

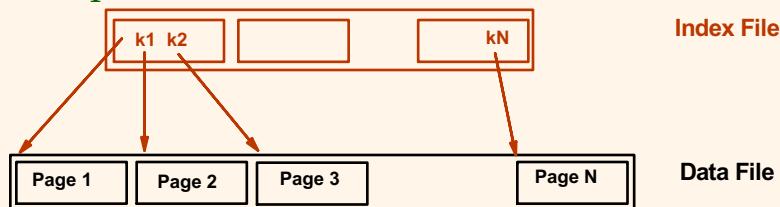


Tree-Structured Indexes: Overview

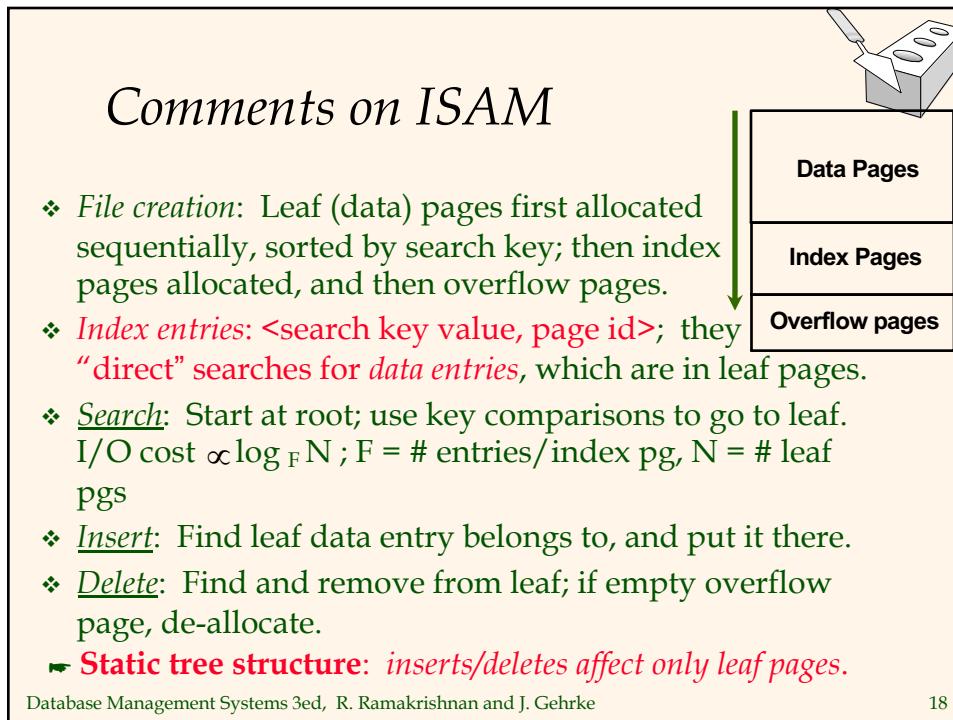
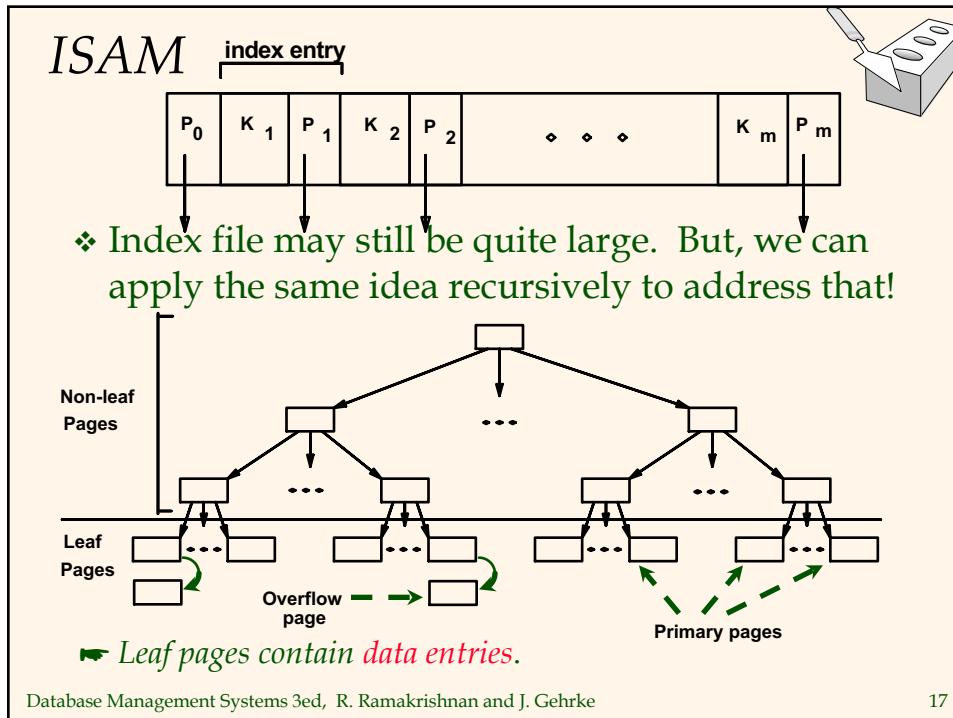
- ❖ As for any index, 3 alternatives for data entries k^* :
 - Data record with key value k
 - $\langle k, \text{rid of data record with search key value } k \rangle$ 使用搜索键值删除数据记录
 - $\langle k, \text{list of rids of data records with search key } k \rangle$ 带有搜索关键字的数据记录的清单列表
- ❖ This data entry choice is orthogonal to the indexing technique used to locate data entries k^* .
- ❖ Tree-structured indexing techniques support both *range searches* and *equality searches*.
- ❖ ISAM: static structure; B+ tree: dynamic, adjusts gracefully under inserts and deletes.

Range Searches

- ❖ “Find all students with $gpa > 3.0$ ”
- ❖ If records are in a sorted file, do binary search to find first such student, then scan to find others.
 - Cost of file binary search can be quite high.
- ❖ Simple idea to do better: add an “index” file.



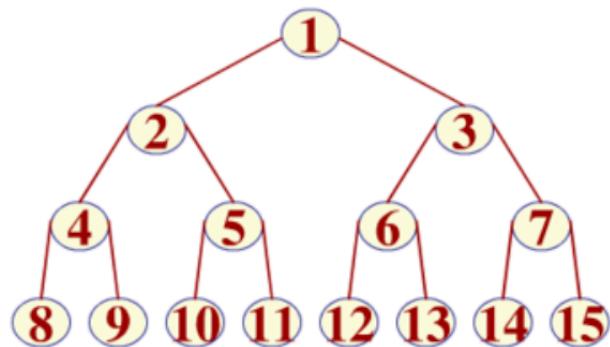
➥ Can now binary search the (smaller) index file!



满二叉树：指的是深度为 k 且含有 2^k-1 个结点的二叉树。

特点：

每一层上的结点数都是最大结点数。



如图，1到15都是结点，8到15是叶子结点，叶子结点就是最大的结点。二叉树就像一棵树，不过这是一棵倒着的树，如图，1是树根，2到7是树杈，8到15是树叶，也就是叶子结点。

B+树在数据库中的应用

{

为什么使用B+树？言简意赅，就是因为：

- 1.文件很大，不可能全部存储在内存中，故要存储到磁盘上
- 2.索引的结构组织要尽量减少查找过程中磁盘I/O的存取次数（为什么使用B-/+Tree，还跟磁盘存取原理有关。）
- 3.局部性原理与磁盘预读，预读的长度一般为页（page）的整倍数，（在许多操作系统中，页得大小通常为4k）
- 4.数据库系统巧妙利用了磁盘预读原理，将一个节点的大小设为等于一个页，这样每个节点只需要一次I/O就可以完全载入，（由于节点中有两个数组，所以地址连续）。而红黑树这种结构，h明显要深的多。由于逻辑上很近的节点（父子）物理上可能很远，无法利用局部性

InnoDB 与 MyISAM 结构上的区别

1.InnoDB的主键索引，MyISAM索引文件和数据文件是分离的，索引文件仅保存数据记录的地址。而在InnoDB中，表数据文件本身就是按B+Tree组织的一个索引结构，这棵树的叶节点data域保存了完整的数据记录。这个索引的key是数据表的主键，因此InnoDB表数据文件本身就是主索引，所以必须有主键，如果没有显示定义，自动为生成一个隐含字段作为主键，这个字段长度为6个字节，类型为长整形

15	18
34	77
Bob	Alice

2.InnoDB的辅助索引（Secondary Index，也就是非主键索引）也会包含主键列，比如名字建立索引，内部节点会包含名字，叶子节点会包含该

Alice	Bob
18	15

名字对应的主键的值，如果主键定义的比较大，其他索引也将很大

3.MyISAM引擎使用B+Tree作为索引结构，索引文件叶节点的data域存放的是数据记录的地址，指向数据文件中对应的值，每个节点只有该索引

15	18
0x07	0x56

列的值

4. **MyISAM主索引和辅助索引（Secondary key）** 在结构上没有任何区别，只是主索引要求key是唯一的，辅助索引可以重复，
(由于MyISAM辅助索引在叶子节点上存储的是数据记录的地址，和主键索引一样，所以相对于B+的InnoDB可通过辅助索引
快速找到所有的数据，而不需要再遍历一边主键索引，所以适用于OLAP)

InnoDB索引和MyISAM索引的区别：

一是主索引的区别，InnoDB的数据文件本身就是索引文件。而MyISAM的索引和数据是分开的。

二是辅助索引的区别：InnoDB的辅助索引data域存储相应记录主键的值而不是地址。而MyISAM的辅助索引和主索引没有多大区别。

}

1. 索引在数据库中的作用

在数据库系统的使用过程当中，数据的查询是使用最频繁的一种数据操作。

最基本的查询算法当然是顺序查找（linear search），遍历表然后逐行匹配行值是否等于待查找的关键字，其时间复杂度为 $O(n)$ 。但时间复杂度为 $O(n)$ 的算法规模小的表，负载轻的数据库，也能有好的性能。但是数据增大的时候，时间复杂度为 $O(n)$ 的算法显然是糟糕的，性能就很快下降了。

好在计算机科学的发展提供了很多更优秀的查找算法，例如二分查找（binary search）、二叉树查找（binary tree search）等。如果稍微分析一下会发现，每种查找算法都只能应用于特定的数据结构之上，例如二分查找要求被检索数据有序，而二叉树查找只能应用于二叉查找树上，但是数据本身的组织结构不可能完全满足各种数据结构（例如，理论上不可能同时将两列都按顺序进行组织），所以，在数据之外，数据库系统还维护着满足特定查找算法的数据结构，这些数据结构以某种方式引用（指向）数据，这样就可以在这些数据结构上实现高级查找算法。这种数据结构，就是索引。

索引是对数据库表中一个或多个列的值进行排序的结构。与在表中搜索所有的行相比，索引用指针指向存储在表中指定列的数据值，然后根据指定的次序排列这些指针，有助于更快地获取信息。通常情况下，只有当经常查询索引列中的数据时，才需要在表上创建索引。索引将占用磁盘空间，并且影响数据更新的速度。但是在多数情况下，索引所带来的数据检索速度优势大大超过它的不足之处。

2. B+树在数据库索引中的应用

目前大部分数据库系统及文件系统都采用B-Tree或其变种B+Tree作为索引结构

1) 在数据库索引的应用

在数据库索引的应用中，B+树按照下列方式进行组织：

① 叶结点的组织方式。B+树的查找键是数据文件的主键，且索引是稠密的。也就是说，叶结点中为数据文件的第一个记录设有一个键、指针对，该数据文件可以按主键排序，也可以不按主键排序；数据文件按主键排序，且B+树是稀疏索引，在叶结点中为数据文件的每一个块设有一个键、指针对；数据文件不按键属性排序，且该属性是B+树的查找键，叶结点中为数据文件里出现的每个属性K设有一个键、指针对，其中指针执行排序键值为K的记录中的第一个。

② 非叶结点的组织方式。B+树中的非叶结点形成了叶结点上的一个多级稀疏索引。每个非叶结点中至少有 $\text{ceil}(m/2)$ 个指针，至多有m个指针。

2) B+树索引的插入和删除

① 在向数据库中插入新的数据时，同时也需要向数据库索引中插入相应的索引键值，则需要向B+树中插入新的键值。即上面我们提到的B-树插入算法。

② 当从数据库中删除数据时，同时也需要从数据库索引中删除相应的索引键值，则需要从B+树中删除该键值。即B-树删除算法

为什么使用B-Tree (B+Tree)

二叉查找树进化的红黑树等数据结构也可以用来实现索引，但是文件系统及数据库系统普遍采用B-/+Tree作为索引结构。

一般来说，索引本身也很大，不可能全部存储在内存中，因此索引往往以索引文件的形式存储在磁盘上。这样的话，索引查找过程中就要产生磁盘I/O消耗，相对于内存存取，I/O存取的消耗要高几个数量级，所以评价一个数据结构作为索引的优劣最重要的指标就是在查找过程中磁盘I/O操作次数的渐进复杂度。换句话说，索引的结构组织要尽量减少查找过程中磁盘I/O的存取次数。为什么使用B-/+Tree，还跟磁盘存取原理有关。

局部性原理与磁盘预读

由于存储介质的特性，磁盘本身存取就比主存慢很多，再加上机械运动耗费，磁盘的存取速度往往是主存的几百分之一，因此为了提高效率，要尽量减少磁盘I/O。为了达到这个目的，磁盘往往不是严格按需读取，而是每次都会预读，即使只需要一个字节，磁盘也会从这个位置开始，顺序向后读取一定长度的数据放入内存。这样做的理论依据是计算机科学中著名的局部性原理：

当一个数据被用到时，其附近的数据也通常会马上被使用。

程序运行期间所需要的数据通常比较集中。

由于磁盘顺序读取的效率很高（不需要寻道时间，只需很少的旋转时间），因此对于具有局部性的程序来说，预读可以提高I/O效率。

预读的长度一般为页（page）的整倍数。页是计算机管理存储器的逻辑块，硬件及操作系统往往将主存和磁盘存储区分割为连续的大小相等的块，每个存储块称为一页（在许多操作系统中，页得大小通常为4K），主存和磁盘以页为单位交换数据。当程序要读取的数据不在主存中时，会触发一个缺页异常，此时系统会向磁盘发出读盘信号，磁盘会找到数据的起始位置并向后连续读取一页或几页载入内存中，然后异常返回，程序继续运行。

我们上面分析B-Tree检索一次最多需要访问节点：

$h =$

$$\log_{\lceil m/2 \rceil} \left(\frac{n+1}{2} \right) + 1$$

数据库系统巧妙利用了磁盘预读原理，将一个节点的大小设为等于一个页，这样每个节点只需要一次I/O就可以完全载入。为了达到这个目的，在实际实现B-Tree还需要使用如下技巧：

每次新建节点时，直接申请一个页的空间，这样就保证一个节点物理上也存储在一个页里，加之计算机存储分配都是按页对齐的，就实现了一个node只需一次I/O。

B-Tree中一次检索最多需要 $h-1$ 次I/O（根节点常驻内存），渐进复杂度为 $O(h) = O(\log m N)$ 。一般实际应用中， m 是非常大的数字，通常超过100，因此 h 非常小（通常不超过3）。

综上所述，用B-Tree作为索引结构效率是非常高的。

而红黑树这种结构， h 明显要深的多。由于逻辑上很近的节点（父子）物理上可能很远，无法利用局部性，所以红黑树的I/O渐进复杂度也为 $O(h)$ ，效率明显比B-Tree差很多。

MySQL的B-Tree索引（技术上说B+Tree）

在 MySQL 中，主要有四种类型的索引，分别为： B-Tree 索引， Hash 索引， Fulltext 索引和 R-Tree 索引。我们主要分析 B-Tree 索引。

B-Tree 索引是 MySQL 数据库中使用最为频繁的索引类型，除了 Archive 存储引擎之外的其他所有的存储引擎都支持 B-Tree 索引。Archive 引擎直到 MySQL 5.1 才支持索引，而且只支持索引单个 AUTO_INCREMENT 列。

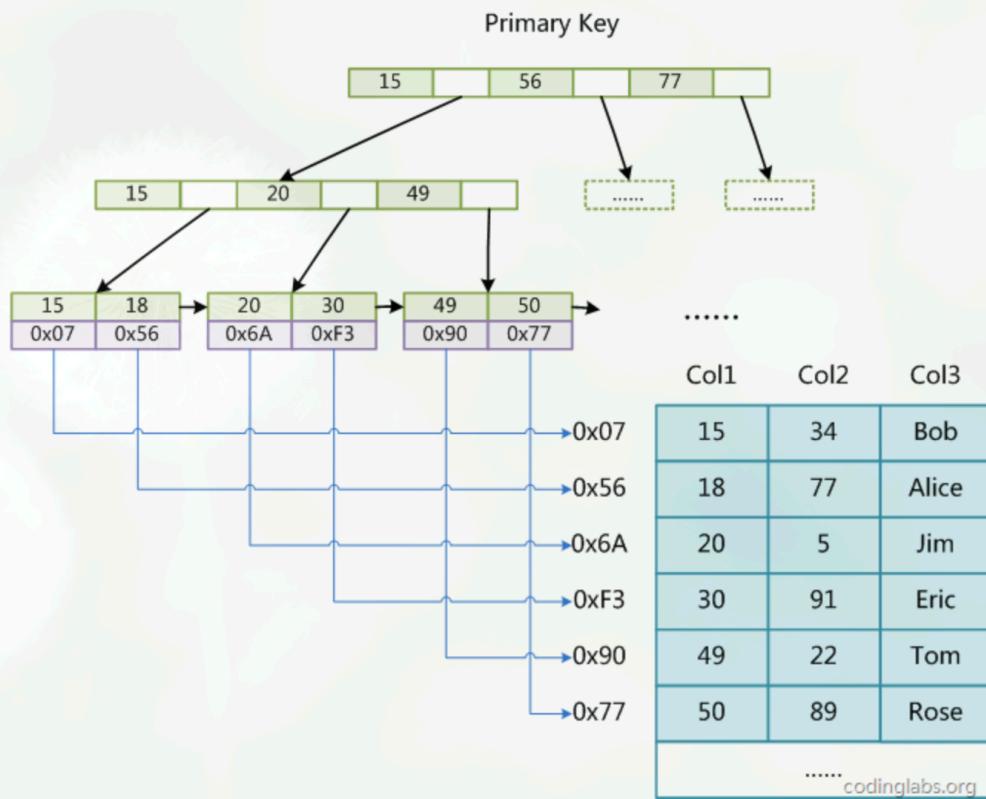
不仅仅在 MySQL 中是如此，实际上在其他的很多数据库管理系统中 B-Tree 索引也同样是作为最主要的索引类型，这主要是因为 B-Tree 索引的存储结构在数据库的数据检索中有非常优异的表现。

一般来说， MySQL 中的 B-Tree 索引的物理文件大多都是以 Balance Tree 的结构来存储的，也就是所有实际需要的数据都存放于 Tree 的 Leaf Node(叶子节点)，而且到任何一个 Leaf Node 的最短路径的长度都是完全相同的，所以我们大家都称之为 B-Tree 索引。当然，可能各种数据库（或 MySQL 的各种存储引擎）在存放自己的 B-Tree 索引的时候会对存储结构稍作改造。如 InnoDB 存储引擎的 B-Tree 索引实际使用的存储结构实际上是 B+Tree，也就是在 B-Tree 数据结构的基础上做了很小的改造，在每一个 Leaf Node 上面除了存放索引键的相关信息之外，还存储了指向与该 Leaf Node 相邻的后一个 LeafNode 的指针信息（增加了顺序访问指针），这主要是为了加快检索多个相邻 Leaf Node 的效率考虑。

1. MyISAM索引实现：

1) 主键索引：

MyISAM引擎使用B+Tree作为索引结构，叶节点的数据域存放的是数据记录的地址。下图是MyISAM主键索引的原理图：

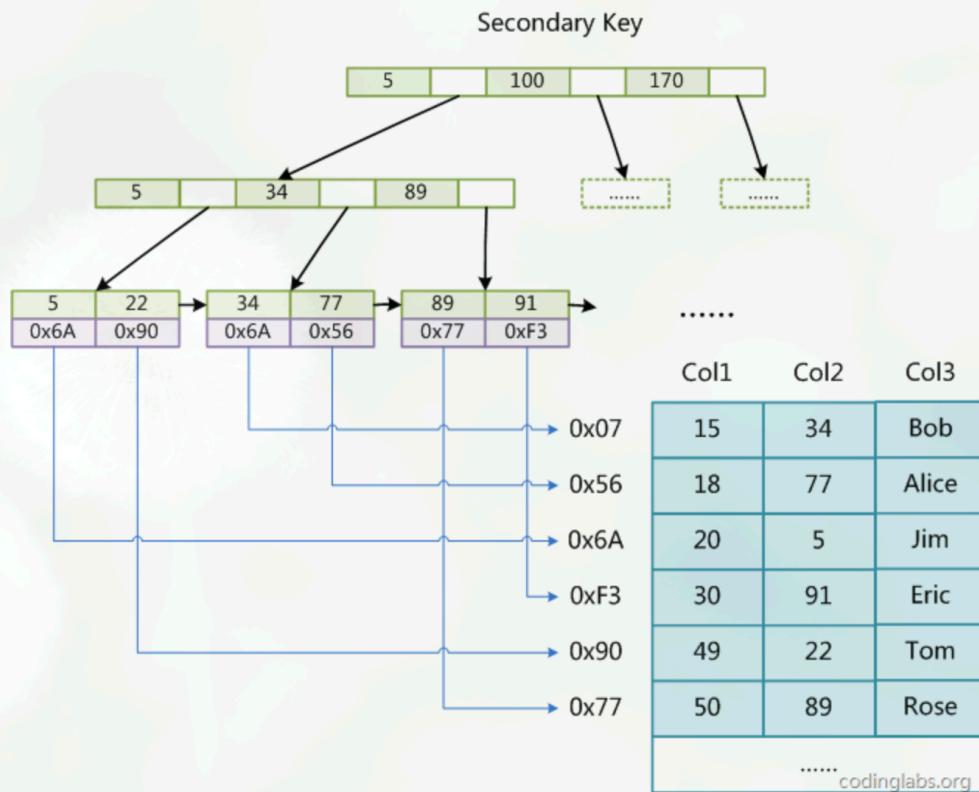


(图myisam1)

这里设表一共有三列，假设我们以Col1为主键，图myisam1是一个MyISAM表的主索引（Primary key）示意。可以看出MyISAM的索引文件仅仅保存数据记录的地址。

2) 辅助索引 (Secondary key)

在MyISAM中，主索引和辅助索引 (Secondary key) 在结构上没有任何区别，只是主索引要求key是唯一的，而辅助索引的key可以重复。如果我们在Col2上建立一个辅助索引，则此索引的结构如下图所示：



同样也是一颗B+Tree，data域保存数据记录的地址。因此，MyISAM中索引检索的算法为首先按照B+Tree搜索算法搜索索引，如果指定的Key存在，则取出其data域的值，然后以data域的值为地址，读取相应数据记录。

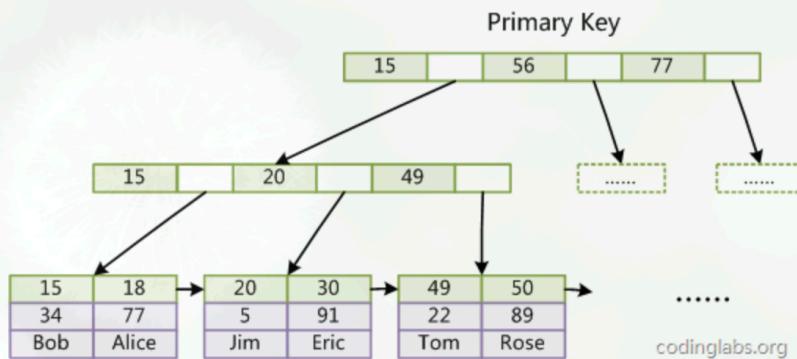
MyISAM的索引方式也叫做“非聚集”的，之所以这么称呼是为了与InnoDB的聚集索引区分。

2. InnoDB索引实现

然InnoDB也使用B+Tree作为索引结构，但具体实现方式却与MyISAM截然不同.

1) 主键索引：

MyISAM索引文件和数据文件是分离的，索引文件仅保存数据记录的地址。而在InnoDB中，表数据文件本身就是按B+Tree组织的一个索引结构，这棵树的叶节点data域保存了完整的数据记录。这个索引的key是数据表的主键，因此InnoDB表数据文件本身就是主索引。

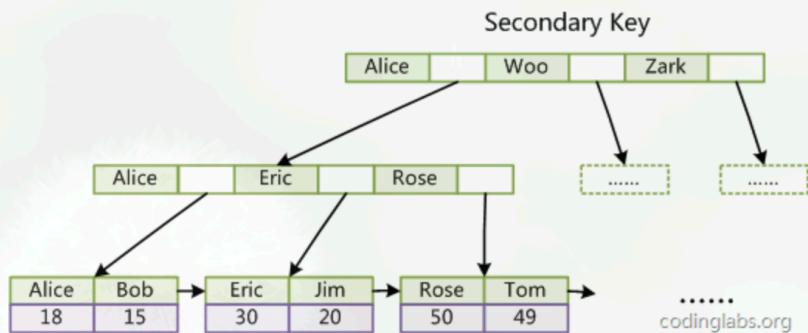


(图inndb主键索引)

(图inndb主键索引) 是InnoDB主索引（同时也是数据文件）的示意图，可以看到叶节点包含了完整的数据记录。这种索引叫做**聚集索引**。因为InnoDB的数据文件本身要按主键聚集，所以InnoDB要求表必须有主键（MyISAM可以没有），如果没有显式指定，则MySQL系统会自动选择一个可以唯一标识数据记录的列作为主键，如果不存在这种列，则MySQL自动为InnoDB表生成一个隐含字段作为主键，这个字段长度为6个字节，类型为长整形。

2) . InnoDB的辅助索引

InnoDB的所有辅助索引都引用主键作为data域。例如，下图为定义在Col3上的一个辅助索引：



InnoDB 表是基于聚簇索引建立的。因此InnoDB 的索引能提供一种非常快速的主键查找性能。不过，它的辅助索引（Secondary Index，也就是非主键索引）也会包含主键列，所以，如果主键定义的比较大，其他索引也将很大。如果想在表上定义、很多索引，则争取尽量把主键定义得小一些。InnoDB 不会压缩索引。

文字符的ASCII码作为比较准则。聚集索引这种实现方式使得按主键的搜索十分高效，但是辅助索引搜索需要检索两遍索引：首先检索辅助索引获得主键，然后用主键到主索引中检索获得记录。

不同存储引擎的索引实现方式对于正确使用和优化索引都非常有帮助，例如知道了InnoDB的索引实现后，就很容易明白为什么不建议使用过长的字段作为主键，因为所有辅助索引都引用主索引，过长的主索引会令辅助索引变得过大。再例如，用非单调的字段作为主键在InnoDB中不是一个好主意，因为InnoDB数据文件本身是一颗B+Tree，非单调的主键会造成在插入新记录时数据文件为了维持B+Tree的特性而频繁的分裂调整，十分低效，而使用自增字段作为主键则是一个很好的选择。

InnoDB索引和MyISAM索引的区别：

一是主索引的区别，InnoDB的数据文件本身就是索引文件。而MyISAM的索引和数据是分开的。

二是辅助索引的区别：InnoDB的辅助索引data域存储相应记录主键的值而不是地址。而MyISAM的辅助索引和主索引没有多大区别。