1. \*\*Clustering Index:\*\*

- \*\*解释：\*\* 聚簇索引是一种数据库索引结构，它决定了物理存储上的数据顺序。与非聚簇索引不同，聚簇索引将索引的键值与实际数据行存储在一起，按照索引键值的顺序在磁盘上组织和存储数据。因此，具有相似索引值的数据行在物理上也会相邻存储，这有助于提高范围查询的性能。

2. \*\*Sequential File Organization:\*\*

- \*\*解释：\*\* 顺序文件组织是一种数据存储结构，其中数据按照某个键值或其他顺序排列。与随机存取不同，顺序文件通常按照逻辑顺序在磁盘或其他存储介质上存储数据。这种组织方式使得在文件中查找、插入或删除记录时可以通过顺序访问实现，而不是直接跳跃到某个位置。这对于需要频繁顺序访问数据的应用场景可能是有效的。

3. \*\*Recoverable Schedule (of transactions):\*\*

- \*\*解释：\*\* 可恢复调度是与数据库事务处理相关的概念，指的是一种安排事务执行的方式，以确保在系统发生故障或中断的情况下，数据库可以被有效地还原到一个一致的状态。在可恢复调度中，一系列事务的执行顺序和操作必须考虑到故障恢复的需求，以保证系统的可靠性和数据的一致性。通过使用事务日志和一致性检查点等机制，可恢复调度可以有效地管理事务的执行和系统的恢复过程。

B+树是一种有效的存储结构，适用于顺序文件组织的原因如下：

1. \*\*范围查询效率高：\*\* B+树是一种平衡树结构，其内部节点存储索引信息，叶子节点存储实际数据。由于B+树的叶子节点形成了一个有序链表，这使得范围查询变得高效。当执行范围查询时，只需遍历叶子节点链表上的部分节点，而无需全局扫描整个数据集。

2. \*\*顺序性与磁盘I/O优化：\*\* B+树的有序性保证了相邻的数据通常在磁盘上也是相邻存储的，这有助于减少磁盘I/O操作的次数。顺序性可以最大程度地利用磁盘预读机制，提高数据的读取效率。

3. \*\*支持范围查询和范围删除：\*\* 由于B+树的叶子节点形成了有序链表，范围查询和范围删除等操作变得更为高效。这种结构使得查找、插入和删除等操作的时间复杂度都相对较低。

B+树的索引和文件组织有以下区别：

1. \*\*B+树索引：\*\* B+树索引是针对数据库表中的某一列或多列数据建立的索引结构。它的目的是提高数据的检索效率，而实际数据仍然存储在表中的数据页上。B+树索引的叶子节点包含指向实际数据行的指针或地址。

2. \*\*B+树文件组织：\*\* B+树文件组织是一种数据存储结构，直接将数据存储在B+树的叶子节点上，而非在独立的数据页上。这种结构使得B+树本身就是一个按顺序组织的文件，而不仅仅是用于索引。查询时可以通过B+树的结构直接找到所需的数据，而无需再访问独立的数据页。

总的来说，B+树索引主要用于加速数据的查找，而B+树文件组织则直接将数据组织在B+树结构中，更适合顺序文件组织的场景。

数据库管理系统提供了抽象数据的概念，通过逻辑模型来隔离物理存储的细节。在这种情况下，Tom和Jerry之间使用的是数据独立性的概念，这是数据库管理系统的一个关键特性。

1. \*\*数据独立性：\*\* 数据库系统通过两种独立性来实现数据抽象，即物理数据独立性和逻辑数据独立性。

- \*\*物理数据独立性：\*\* 允许数据库管理员在不影响应用程序的前提下，改变数据的物理存储结构。Jerry改变了物理组织方式，而Tom在使用逻辑模型时不受影响。这使得数据库管理员可以优化性能、进行备份和维护等操作，而不会对应用程序的逻辑产生直接影响。

- \*\*逻辑数据独立性：\*\* 允许应用程序的逻辑模型（也称为逻辑架构或模式）独立于底层物理存储细节。Tom在继续工作时无需知道Jerry对数据的物理调整，因为他操作的是逻辑数据模型。这使得应用程序的开发者能够专注于逻辑层面的设计和实现，而不必关心底层的物理存储细节。

通过这两种数据独立性的支持，数据库管理系统确保了数据库的灵活性和可维护性，使得开发人员和管理员可以在不同的层面上独立地工作，提高了数据库系统的整体效率和可维护性。

有人可能认为在数据库事务管理中，数据一致性要求总是应该被满足的。然而，有时候弱一致性级别在实际应用中是有用的，特别是在需要权衡一致性和性能的场景。

\*\*例子：\*\*

考虑一个在线社交媒体平台，用户可以发布状态更新并查看其他用户的状态。在这个场景中，如果所有的读操作都必须等待最新的写操作完成，可能会导致系统性能下降。在这种情况下，使用较弱的一致性级别可能是合理的。

\*\*弱一致性级别的用处：\*\*

1. \*\*读未提交（Read Uncommitted）：\*\* 允许一个事务读取另一个事务未提交的数据。在上述社交媒体平台的例子中，一个用户发布状态更新后，其他用户可以立即看到未提交的状态，从而提供了更快的响应时间。尽管这可能导致读取到未提交的、不一致的数据，但在某些场景下，这种权衡是可以接受的，特别是在强调实时性和性能的应用中。

总体而言，弱一致性级别可以在需要牺牲一些一致性的情况下提高系统的性能和吞吐量。然而，在一些对数据一致性要求非常高的场景，如金融交易等，强一致性仍然是更为合适的选择。选择适当的一致性级别通常取决于具体应用的需求和权衡。

SELECT A,D,E FROM r,s WHERE C=D AND B=0 AND E>1000



SELECT A,B,C FROM r WHERE C IN (SELECT C FROM s WHERE E =0)

UPDATE book

SET on\_shelf\_count = total\_count -

(SELECT COUNT(\*)

FROM rent\_record\_detail

WHERE book\_id = book.book\_id

AND book\_status = 'IN\_RENT');

(iii) \*\*触发器的部署及其利弊：\*\*

如果要通过触发器维护 `on\_shelf\_count`，可能需要在 `rent\_record\_detail` 表上部署触发器。以下是可能的触发器：

- 当有新的租借记录插入时，触发器减少相应书的 `on\_shelf\_count`。

- 当租借记录中的书被归还时，触发器增加相应书的 `on\_shelf\_count`。

\*\*利弊讨论：\*\*

\*\*优势：\*\*

- \*\*自动维护：\*\* 触发器可以自动执行，无需手动干预，确保 `on\_shelf\_count` 始终是最新的。

- \*\*数据一致性：\*\* 触发器可以保证 `on\_shelf\_count` 与实际情况一致，减少了错误的可能性。

\*\*劣势：\*\*

- \*\*性能开销：\*\* 触发器的执行可能引入性能开销，尤其是在数据量很大的情况下。

- \*\*复杂性：\*\* 触发器可能增加系统的复杂性，需要确保它们的执行不会导致死锁或其他并发问题。

- \*\*维护难度：\*\* 在某些情况下，维护触发器可能会变得复杂，尤其是当系统逐渐演变时，可能需要对触发器进行修改或扩展。

对的。对BD进行闭包运算可以得到ABCDE全部。

删除具有特定D值的元组，那么与该D值相关的其他信息也丢失了。

求最小函数依赖集  
一步步来，先拆为ABC->D ,~~ABC->E~~, B->C, ~~AB->C~~, AB->E, D->A

若没有ABC->D, ABC->E，保留

若没有ABC->E,ABC->D,E，说明ABC->E多余了，（因为能被剩下的关系推出）,去除。

若没有B->C, 则B没法推出C了，保留 B->C

若没有AB->C, 那么AB->E,C，（D），说明AB->C多余了，去除,

若没有AB->E, 则AB->C,D, 保留。

若没有D->A, 则D推不出A了，保留。

现在有ABC->D,B->C, AB->E,D->A。

B->C,所以AB->D即可。答案是AB->D,B->C,AB->E,D->A。

由上可得左边出现的A,B,D，右边出现的D,C,E,A。去掉重复的A,D。

B能推出全部么？不能。加上A呢？可以。所以AB是候选键。

第三范式：R1(ABDE),R2(BC),R3(DA). AB已在R1中出现，所以是无损连接。

其他题目中，若AB不在上述关系中，则再单独开一个关系R(AB)即可。

BC范式——所有依赖项的左边都必须是候选码。

球员表现排名查询: 列出在一个给定赛季中得分最多的球员。

比赛统计分析: 统计每场比赛中球员的助攻、犯规等技术统计数据。

球队对比查询: 比较两支球队在特定比赛类型中的得分情况，以便战术和策略的调整

1. \*\*Player（球员）:\*\*

- \*\*属性:\*\*

- PlayerID (主键): 球员的唯一标识符。

- Name: 球员姓名。

- DateOfBirth: 球员出生日期。

- Age: 球员年龄。

- Height: 球员身高。

- Weight: 球员体重。

- Phone: 球员联系电话。

- \*\*关系:\*\*

- ParticipatesIn（参与比赛关系）: 通过联合主键（PlayerID, MatchID）与 Match 实体关联，表示球员参与的比赛，还包括球员是否是队长（IsCaptain）的信息。

- LivesAt（居住地关系）: 通过主键（PlayerID）与 Player 实体关联，表示球员的居住地信息，包括 HomeAddress、Province、City 和 DetailedAddress。

2. \*\*Match（比赛）:\*\*

- \*\*属性:\*\*

- MatchID (主键): 比赛的唯一标识符。

- Type: 比赛类型（例如，联赛、俱乐部杯、洲际杯等）。

- Location: 比赛地点。

- Time: 比赛时间。

- TeamTotalScore: 球队总得分。

- OpponentTeamName: 对手球队名称。

- OpponentTeamScore: 对手球队得分。

- \*\*关系:\*\*

- ParticipatesIn（参与比赛关系）: 通过联合主键（PlayerID, MatchID）与 Player 实体关联，表示球员参与的比赛。

3. \*\*Performance（比赛表现）:\*\*

- \*\*属性:\*\*

- PerformanceID (主键): 表现的唯一标识符。

- Goals: 投篮命中次数。

- Assists: 助攻次数。

- Faults: 犯规次数。

- TimePlayed: 上场时间。

- \*\*关系:\*\*

- 可以与 Match 实体关联，表示比赛表现的信息。

4. \*\*ParticipatesIn（参与比赛关系）:\*\*

- \*\*属性:\*\*

- PlayerID (联合主键): 参与比赛的球员的唯一标识符。

- MatchID (联合主键): 参与的比赛的唯一标识符。

- IsCaptain: 标志球员是否是队长。

- \*\*关系:\*\*

- 与 Player 和 Match 实体关联，表示球员参与的比赛，包括是否是队长的信息。

5. \*\*LivesAt（居住地关系）:\*\*

- \*\*属性:\*\*

- PlayerID (主键): 球员的唯一标识符。

- HomeAddress: 家庭地址。

- Province: 居住省份。

- City: 居住城市。

- DetailedAddress: 详细地址。

- \*\*关系:\*\*

- 与 Player 实体关联，表示球员的居住地信息。

SELECT Player.Name, COUNT(ParticipatesIn.MatchID) AS MatchesPlayed, AVG(Performance.Goals) AS AvgGoals

FROM Player

JOIN ParticipatesIn ON Player.PlayerID = ParticipatesIn.PlayerID

JOIN Performance ON ParticipatesIn.MatchID = Performance.MatchID

WHERE YEAR(Match.Time) = 2020

GROUP BY Player.PlayerID, Player.Name;

球员跨队转会处理: 在 ParticipatesIn 关系中，可能需要添加球员在每个赛季中所属的队伍信息，以便跟踪球员的跨队转会。

球员退役记录: 可以添加一个新的关系 Retirement(PlayerID, RetirementDate)，以记录球员的退役信息。

球员交易历史: 如果要追踪球员的交易历史，可以考虑添加一个新的关系 Transfer(PlayerID, FromTeam, ToTeam, TransferDate)。