我的实现方法如下：

（我实现这个程序的目的是与 MySQL 的结果作对照，以保证结果的正确性，因此很多地方没有做过多的优化。）

数据以原始方式（csv）存储。当表较小时，数据存储在内存里（vector<string>），每个表最多在内存中存储 1,000,000 行。超过这一限制后，所有数据被写入文件中，每个文件最多存储 5,000,000 行。

为所有在 WHERE 子句中有“=”条件的列建立索引（包括 join condition 和 match predicate）。索引全部保存在内存中，类型为 unordered_map<string, int>。因为表与表之间的 join 关系是非常有限的（通常表现为具有 foreign key 关系），同时 match predicate 又不是特别多，所以内存可以容纳这些索引。索引的 key 为列值的文本形式，整数也以字符串形式存储；value 为行号（每个表中的数据始终维持插入的顺序），若数据存储在内存中，行号即为下标，若数据存储在文件中，通过一个索引（数组）将行号转换为文件中的偏移量（通过取模运算可以得到文件的 id）。

另外，对于类型为整数的列，我维护了最大最小值，以便查询优化时使用。

关于查询优化，最重要的就是查询树的形状和 join 执行的顺序。查询树的形状可以有多种，我的程序在执行时，会从起始表开始，一个表一个表地做 join，换句话说，查询树是一个链。关于 join 执行的顺序，正常的方法是估计每次 join 的中间结果大小，结合 join 的算法，给出一定的代价。我是根据每个表的 predicate，得出每个表中有多大比例的行会被选出，然后用比例最小的表作为起始表。其它表的 join 顺序是从由起始表开始的 BFS 序中选出字典序最小的一个。注意我并没有用上面所说的比例乘以表的行数，因为有些表虽然很小，但与其它表做 join 之后可能会产生很大的中间结果。

执行查询时，如果起始表上有索引，则使用 distinct value 数量最多的一个，否则顺序地扫描，同时用其它条件过滤。中间结果与其它表做 join 时，全部使用索引，同时用该表的条件过滤。最后调整结果的列的顺序。

（上面的方法存在许多不足之处，请大家批判地来看待。）