微信全文搜索优化之路

jiaminchen

本文首次发表在《程序员》杂志 2017 年 09 月期。

前言

基于本地数据的全文搜索（Full-Text-Search，FTS）在移动应用上扮演着重要的角色。与基于服务端提供的搜索服务不同，移动端受硬件条件限制，尤其在数据量相对较大的情况下，搜索性能问题表现得十分突出。本文以移动平台广泛采用的SQLite FTS Extension为例，介绍了移动平台FTS的基本原理，结合微信安卓客户端自身实践，重点讲述微信在FTS上的一些性能优化经验。

SQLite FTS Extension

SQLite FTS Extension是SQLite为全文搜索开发的一个插件，它是内嵌在标准的SQLite分布版本当中，它具有如下的特点：

搜索速度快：使用倒排索引加速查找过程

稳定性好：目前SQLite在移动端的稳定性比较好，FTS Extension就是SQLite的基础上搭建的

接入简单：Android和IOS平台本身就支持SQLite，并且FTS Extension的使用就和正常使用SQLite表一样。

兼容性好：受益于SQLite本身兼容性很好，SQLite FTS Extension也有很好的兼容性。

目前SQLiteFTSExtension发布了5个版本，我简单说下三个主流的版本。

FTS3：基础版本，具有完整的FTS特性，支持自定义分词器，库函数包括Offsets，Snippet。

FTS4：在FTS3的基础上，性能有较大优化，增加相关性函数计算MatchInfo。

FTS5：和FTS4有较大变动，储存格式上有较大改进，最明显就是Instance-List的分段存储，能够支持更大的Instance-List的存储；并且开放ExtensionApi，支持自定义辅助函数。FTS5发布于2015年中。

存储架构

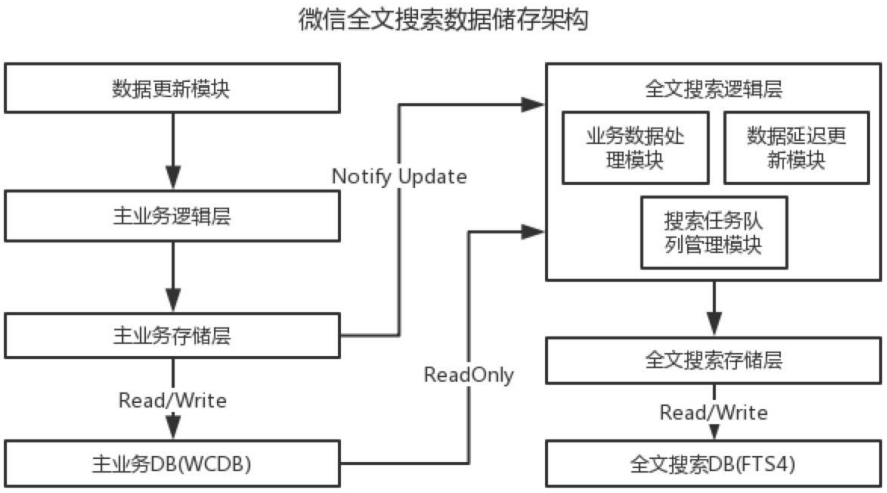
微信全文搜索在2014 年底上线，最初主要服务于联系人和聊天记录的业务搜索。在方案设计之初，为了让这个功能有很好的体验，同时考虑到未来接入业务的会不断增多，我们设计目标是：

1. 搜索速度快

微信全文搜索使用SQLite FTS4 Extension，通过倒排索引提高搜索速度。

2. 业务独立性

微信的核心业务是联系人和消息，而微信全文搜索无论是在建立索引、更新索引或者删除索引时，都需要处理大量数据，为了使得全文搜索不影响微信的核心业务，采用如下的存储架构：



独立DB、读写分离：微信全文搜索在整体架构上独立于主业务，搜索DB也是独立于主业务DB；当主业务数据发生更新时，主业务通过EventBus方式通知搜索对应的业务数据处理模块，业务数据处理模块会通过一个独立的ReadOnly数据库连接接访问主业务数据库，不和主业务存储层共享数据库连接。

减少数据库操作：在搜索模块中，会有专门处理业务数据的模块，对一些复杂的数据结构做一些特殊的处理。例如对于一个500成员的群聊，如果把500个群成员分次插入搜索DB当中，会造成过多的数据库操作。所以，微信会把所有的群成员拼接为单个字符串，插入搜索DB中。

热数据延迟更新： 针对更新频率非常高的热数据，采用延迟更新的策略。所有的索引数据分为正常数据和脏数据。当数据发生更新时，先把对应的数据标记为脏数据，然后有一个定时器，每隔10分钟，把数据更新到索引中。

3. 可扩展性高

高可扩展性要求搜索表结构和业务解耦。SQLite FTS官网上的例子，都是以单索引表的方式，每一列对应业务的某一个属性，当对应业务发生变化，需要修改索引表的结构。为了解决业务变化而带来的表结构修改问题，微信把业务属性数字化，设计如下的表结构：

索引表-IndexTable

ColumnName	ColumnType	Comment
DocId	Long	索引表自动生成的ID，RowId的别名
Content	String	索引字段（每当插入一条Content，FTS会将它分词，插入索引）

数据表-MetaTable

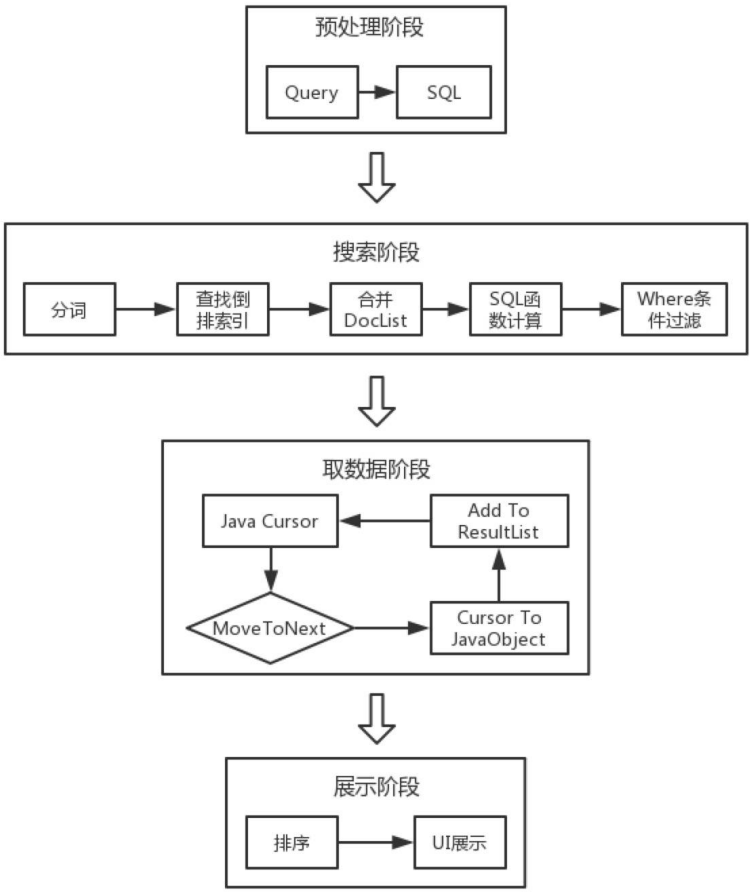
ColumnName	ColumnType	Comment
DocId	Long	外链ID（IndexTable DocId）
Type	Integer	业务主类型
SubType	Integer	业务子类型
BusItemId	String	业务ItemID
TimeStamp	Long	时间戳
Status	Integer	数据状态

IndexTable负责全文搜索的索引建立，它和逻辑无关，当搜索关键词时，只需要找到对应的DocId即可。MetaTable负责业务逻辑的过滤，通过Type和SubType来过滤对应业务的数据，最后输出BusItemId。

搜索优化

微信全文搜索于2014年1月26日5.4版本上线，到2017年春节后的6.5.7版本，总体用户量从4亿增加到9亿，重度用户数量也大幅度增长，微信本地搜索的数据量也大幅度增长，造成了搜索速度不断下降，用户投诉不断增加。我们统计过，从微信5.4版本到6.5.7版本，微信全文搜索各个任务的平均搜索时间增长超过10倍，给微信全文搜索带来巨大挑战。

为了优化搜索时长，先看下搜索的流程图：



通过每个阶段的耗时，发现在取数据阶段，时间占比达到80%以上，并且搜索的结果集数据量越大，时间占比越高，最高可以达到95%。取数据阶段是一个循环的过程，所以优化一个循环需要从两方面着手，减少单次循环耗时和减少总体循环次数。

减少单次循环执行耗时

深入SQLite FTS4 Extension源码，发现FTS4的库函数Offsets耗时占单次循环执行耗时70%以上，并且数据量越大耗时越长。

FTS4库函数Offsets：用于把词语偏移转为字节偏移，微信当中使用字节做结果排序和结果高亮。

函数输入：

- Query：用户查找的关键词
- 命中Doc：关键词所命中的文档。文档就是全文搜索中的基本单位，可以是一个网页，一篇文章或者是一条聊天记录
- 目标词语偏移：在搜索阶段，通过关键词查找搜索索引可以拿到目标词语偏移

函数输出：

- 目标字节偏移：表示关键词在命中Doc中的字节偏移。

例如：

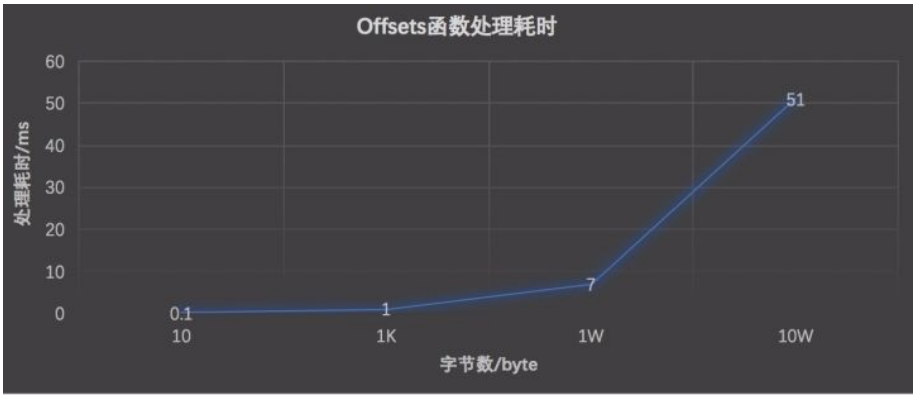
Query=我 命中Doc=我和我弟弟去逛街 目标词语偏移=0、2

把命中Doc经过分词器分词，可以得到下表：

句子	我	和	我	弟	弟	去	逛	街
词语偏移	0	1	2	3	4	5	6	7
字节偏移	0	3	6	9	12	15	18	21

最后计算可以得出目标字节偏移=0、6

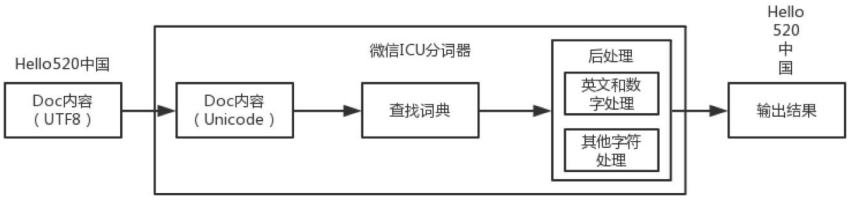
下图是Offsets函数处理命中Doc字节数和耗时的关系：



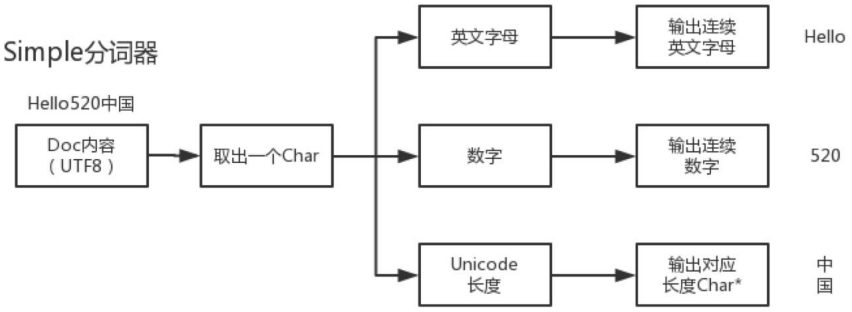
Offsets函数的处理过程中包括分词，所以第一步就优化分词器。

要优化分词器，分词规则是重中之重。微信的分词规则为英文和数字合并分词，非英文和数字单独分词。举个例子，如对于昵称“Hello520中国”，分词结果为“Hello”、“520”、“中”、“国”。这个分词规则的原因主要是在微信对全文搜索的结果排序需求主要是其他的属性排序，并非依据文档的相关性排序。即，全文搜索部分只需要找到存在关键词的文档，并不关心文档中存在几个关键词。而且用户的输入Query大部分情况都不能组成词语，存在方言，所以把整个词语全部拆开建立索引是符合需求的。

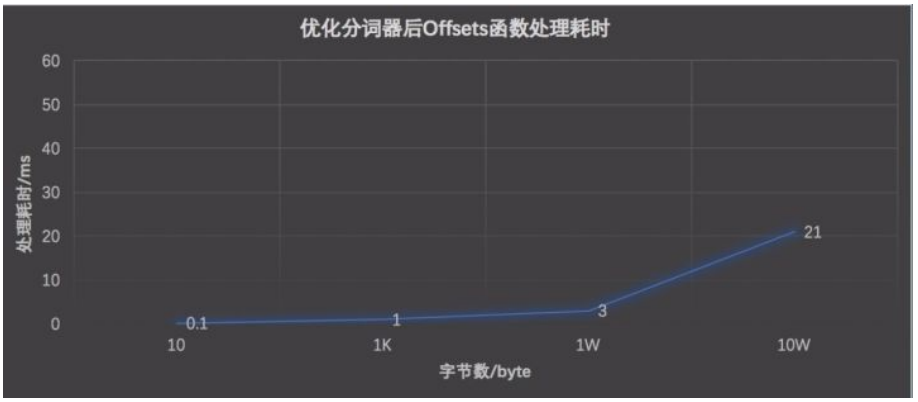
微信全文搜索最早开发于2013年底，FTS4是SQLite FTS Extension的最高版本，但是FTS4自带的分词器不能很好的支持中文，只能使用ICU分词器，当时ICU分词器的接入比较简单，对中文支持较好，所以使用了ICU分词器。



对于昵称“Hello520中国”输出分词器中，开始是UTF8编码，分词器会做一次转化为Unicode编码，接着查找词典，最后进行后处理得到分词结果。从输入输出中可以发现，转化编码和查找词典这两步其实是多余的，所以微信舍弃ICU分词器，自定义了Simple分词器。



Simple分词直接处理的UTF8编码的Doc内容，通过单个char，判断当前字符的Unicode编码范围和Unicode编码长度，根据不同的情况做出不同的处理。



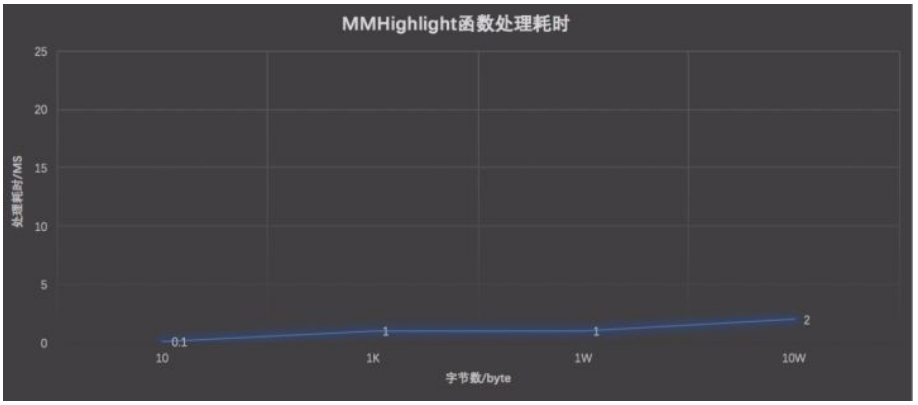
经过分词器优化后Offsets函数耗时在处理10万Byte的耗时降低为21ms，但是这样的优化还不够，当处理超过10个10W结果Doc时，仍然会超过200ms，所以有了下一步的优化。

在移动端由于屏幕的限制，往往在最后显示搜索结果时，只会高亮少量命中的关键词，而Offsets函数会计算命中Doc中所有目标词语偏移，所以需要对Offsets函数进行改造。

最开始我尝试的方案是直接修改Offsets函数源码，发现FTS4对API的封装比较难使用，Offsets函数的依赖也比较多，修改出来的代码很难维护，可读性也不好，所以需要寻找新的方法来优化。在一番研究以后，我发现FTS5支持自定义辅助函数，并且有比较好的API的封装，所以最后使用FTS5自定义辅助函数(MMHighLight)重新实现Offsets函数的功能，并加入优化逻辑。

输入：Query=我 命中Doc=我和我弟弟去逛街 目标词语偏移=0、2 目标返回个数=1

分词器分步回调，当分词器第一次返回“我”，符合目标词语偏移的第一个0，并且此时已经满足目标返回个数1个，函数直接返回目标字节偏移=0。



减少总体循环次数

减少取数据阶段的总体循环次数，比较容易想到的就是在SQL层做数据的分页返回，分页返回就意味着需要在DB层排序，在DB层排序的决定因素就是排序因子。但是微信全文搜索面对的业务排序因子多并且复杂，无法直接使用SQL中的ORDER BY，所以需要通过一个中间函数转化，把所有的排序因子通过一个可比较的数字体现，最后再使用ORDER BY排序。

这里简单说下，比较复杂的排序因子如下：

时间分段排序：时间范围在半年内，排序因子取决于下一级排序因子，时间范围在半年外，取决于时间的远近。

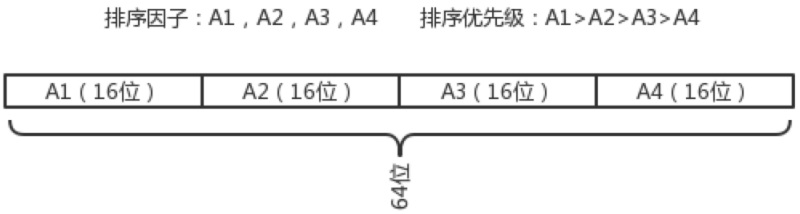
函数结果排序：排序因子是一个函数计算的结果，不是一个直接的数据库Column，并且函数计算结果不可直接使用ORDER BY，例如字符串形式的数字。

通过以上的分析，减少总体循环次数的核心点就在于，把Java层的排序转移到SQL层去做，优点如下：

1. 减少I/O

2. 减少C层到Java层的数据拷贝

所以这里关键的实现点在于中间转化函数的实现，微信的中间转化函数MMRank是通过FTS5的辅助函数实现的。



MMRank的实现原理就是通过把所有的排序因子转化到一个64位的Long数值当中，高优先级的排序因子置高位，低优先级的排序因子置低位。最后的SQL如下：

```
SELECT MMRank(A1, A2, A3, A4) AS Rank FROM
IndexTable ORDER BY RANK DESC;
```

特殊优化——聊天记录搜索优化

微信全文搜索中有一个比较特殊的搜索任务，就是聊天记录。

如图所示：

图中的红色圈内的数字表示，此会话中，包含关键字“我”的聊天记录的个数，而会话的排序规则就是会话的活跃时间。

微信聊天记录的搜索有一下两个特点：

1. 有统计属性
2. 数量非常多（单关键词命中最高可达到20万条）

从搜索流程图中可以看出，微信最初采用的方案是在Java层统计个数和排序，此方法在大数据的情况下不可取。鉴于之前分析过减少循环次数可以通过分页返回，其核心点在于把排序从Java层转移到SQL层，所以就有了优化方案一。

优化方案一：Group By

实现SQL如下：

```
SELECT count(conv), MAX(timestamp) AS MaxTime
FROM IndexTable GROUP BY conv ORDER BY MaxTime
DESC LIMIT 4;
```

此方案通过Group By在SQL层直接统计出命中聊天记录的个数，并按照最近的时间排序，但是也有明显的缺陷：

1. 无法使用索引加速：当GroupBy和OrderBy同时使用是，OrderBy中必须包含GroupBy的字段才可以命中索引，原因是使用GroupBy会生成中间子表。
2. 全量计算：GroupBy在SQL层统计命中聊天记录个数是统计了所有会话，上图中只需要统计3个会话，浪费了大量资源。

优化方案二：分步计算

鉴于方案一全量计算的问题，采用分步计算的方式。

第一步：找出最近活跃的3个会话

```
SELECT count(*) FROM IndexTable ORDER BY
timestamp DESC LIMIT 3;
```

得到会话conv1,conv2,conv3，然后执行以下SQL，可以分别得到三个会话的命中个数

```
SELECT count(*) FROM IndexTable WHERE
conv='conv1';
SELECT count(*) FROM IndexTable WHERE
conv='conv2';
SELECT count(*) FROM IndexTable WHERE
conv='conv3';
```

但是这种方法也存在问题，需要执行多条SQL。

优化方案三：MessageCount

鉴于方案二需要多条SQL的问题，可以通过自定义聚合函数实现一次性统计。执行步骤如下：

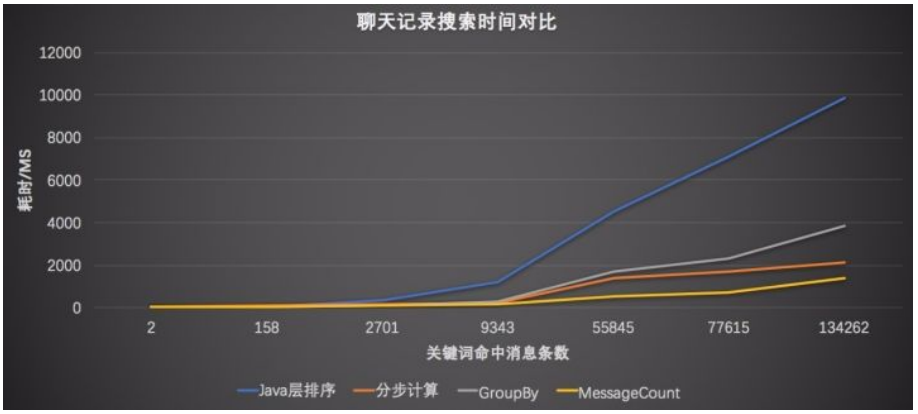
第一步：找出最近活跃的3个会话

```
SELECT count(*) FROM IndexTable ORDER BY
timestamp DESC LIMIT 3;
```

得到会话conv1,conv2,conv3，然后执行以下SQL

```
SELECT MessageCount(3) FROM IndexTable WHERE
conv
IN ('conv1','conv2','conv3');
```

可以一次性得到三个会话的命中个数。



最后

经过优化后，微信全文搜索全体用户各个任务平均耗时都在50ms以下，而重度用户各个任务的平均搜索耗时都在200ms以下，平均时间优化的幅度达到5倍以上。

后续还有很多值得优化的地方，例如，在计算高亮时，如果在DocList的数据结构中，直接加入字节偏移，那么还可以节省一部分时间。

最后希望我的分享能够对大家有些价值，欢迎留言交流。