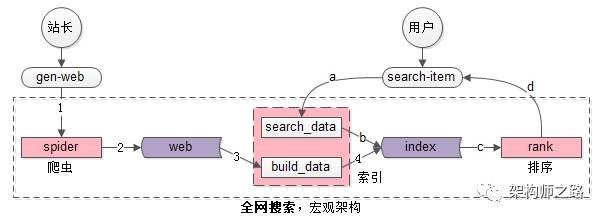
**搜索架构**

1. **搜索引擎的架构，原理与细节**
   1. ***全网搜索引擎架构与流程***



全网搜索的宏观架构如上图，主要分为三部分(粉色)

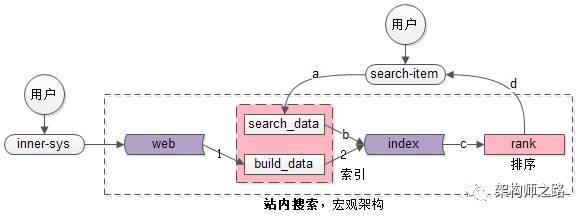
* spider爬虫系统
* search&index建立索引与查询索引系统，这个系统又主要分为两部分：一部分用于生成索引数据build\_index，一部分用于查询索引数据search\_index
* rank打分排序系统

核心数据主要分为两部分(紫色)

* web网页库
* index索引数据

全网搜索引擎的业务特点决定了，这是一个“写入”和“检索”完全分离的系统：

* 写入
  + 系统组成：由spider与search&index两个系统完成
  + 输入：站长们生成的互联网网页
  + 输出：正排倒排索引数据
  + 流程：如架构图中的1，2，3，4
    - spider把互联网网页抓过来
    - spider把互联网网页存储到网页库中（这个对存储的要求很高，要存储几乎整个“万维网”的镜像）
    - build\_index从网页库中读取数据，完成分词
    - build\_index生成倒排索引
* 【检索】
  + 系统组成：由search&index与rank两个系统完成
  + 输入：用户的搜索词
  + 输出：排好序的第一页检索结果
  + 流程：如架构图中的a，b，c，d
    - search\_index获得用户的搜索词，完成分词
    - search\_index查询倒排索引，获得“字符匹配”网页，这是初筛的结果
    - rank对初筛的结果进行打分排序
    - rank对排序后的第一页结果返
  1. ***站内搜索引擎架构与流程***



站内搜索引擎的宏观架构如上图，与全网搜索引擎的宏观架构相比，差异只有写入的地方：

* 全网搜索需要spider要被动去抓取数据
* 站内搜索是内部系统生成的数据，例如“发布系统”会将生成的帖子主动推给build\_data系统

看似“很小”的差异，架构实现上难度却差很多：全网搜索如何“实时”发现“全量”的网页是非常困难的，而站内搜索容易实时得到全部数据。

对于spider、search&index、rank三个系统：

* spider和search&index是相对工程的系统
* rank是和业务、策略紧密、算法相关的系统，搜索体验的差异主要在此，而业务、策略的优化是需要时间积累的，这里的启示是：
  + Google的体验比Baidu好，根本在于前者rank牛逼
  + 国内互联网公司（例如360）短时间要搞一个体验超越Baidu的搜索引擎，是很难的，真心需要时间的积累
  1. ***搜索原理与核心数据结构***
     1. *正排索引*

由key查询实体的过程，是正排索引。

* + 用户表：t\_user(uid, name, passwd, age, sex)，由uid查询整行的过程，就是正排索引查询。
  + 网页库：t\_web\_page(url, page\_content)，由url查询整个网页的过程，也是正排索引查询。

网页内容分词后，page\_content会对应一个分词后的集合list<item>。

简易的，正排索引可以理解为Map<url, list<item>>，能够由网页快速（时间复杂度O(1)）找到内容的一个数据结构。

* + 1. *倒排索引*

由item查询key的过程，是倒排索引。

对于网页搜索，倒排索引可以理解为Map<item, list<url>>，能够由查询词快速（时间复杂度O(1)）找到包含这个查询词的网页的数据结构。

举个例子，假设有3个网页：

url1 -> “我爱北京”

url2 -> “我爱到家”

url3 -> “到家美好”

这是一个正排索引Map<url, page\_content>。

分词之后：

url1 -> {我，爱，北京}

url2 -> {我，爱，到家}

url3 -> {到家，美好}

这是一个分词后的正排索引Map<url, list<item>>。

分词后倒排索引：

我 -> {url1, url2}

爱 -> {url1, url2}

北京 -> {url1}

到家 -> {url2, url3}

美好 -> {url3}

由检索词item快速找到包含这个查询词的网页Map<item, list<url>>就是倒排索引。

正排索引和倒排索引是spider和build\_index系统提前建立好的数据结构，为什么要使用这两种数据结构，是因为它能够快速的实现“用户网页检索”需求（业务需求决定架构实现）。

* + 1. *搜索的过程*

假设搜索词是“我爱”

* + 分词，“我爱”会分词为{我，爱}，时间复杂度为O(1)
  + 每个分词后的item，从倒排索引查询包含这个item的网页list<url>，时间复杂度也是O(1)：

我 -> {url1, url2}

爱 -> {url1, url2}

* + 求list<url>的交集，就是符合所有查询词的结果网页，对于这个例子，{url1, url2}就是最终的查询结果

看似到这里就结束了，其实不然，分词和倒排查询时间复杂度都是O(1)，整个搜索的时间复杂度取决于“求list<url>的交集”，问题转化为了求两个集合交集。

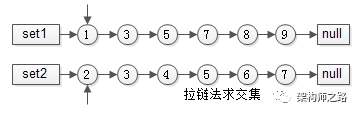
字符型的url不利于存储与计算，一般来说每个url会有一个数值型的url\_id来标识，后文为了方便描述，list<url>统一用list<url\_id>替代。

list1和list2，求交集的方案

* + ***for \* for，土办法，时间复杂度O(n\*n)***

每个搜索词命中的网页是很多的，O(n\*n)的复杂度是明显不能接受的。倒排索引是在创建之初可以进行排序预处理，问题转化成两个有序的list求交集，就方便多了。

* + ***有序list求交集，拉链法***



有序集合1{1,3,5,7,8,9}和有序集合2{2,3,4,5,6,7}两个指针指向首元素，比较元素的大小：

* + - 如果相同，放入结果集，随意移动一个指针
    - 否则，移动值较小的一个指针，直到队尾

这种方法的优点在于：

* + - 集合中的元素最多被比较一次，时间复杂度为O(n)
    - 多个有序集合可以同时进行，这适用于多个分词的item求url\_id交集
  + ***分桶并行优化***

数据量大时，url\_id分桶水平切分+并行运算是一种常见的优化方法，如果能将list1<url\_id>和list2<url\_id>分成若干个桶区间，每个区间利用多线程并行求交集，各个线程结果集的并集，作为最终的结果集，能够大大的减少执行时间。

举例：

有序集合1{1,3,5,7,8,9,10,30,50,70,80,90}和有序集合2{2,3,4,5,6,7, 20,30,40,50,60,70}求交集，先进行分桶拆分：桶1的范围为[1, 9]，桶2的范围为[10, 100]，桶3的范围为[101, max\_int]

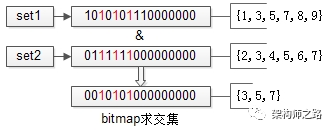
于是：集合1就拆分成：集合a{1,3,5,7,8,9}，集合b{10,30,50,70,80,90}，集合c{}；集合2就拆分成：集合d{2,3,4,5,6,7}，集合e{20,30,40,50,60,70}，集合e{}

每个桶内的数据量大大降低了，并且每个桶内没有重复元素，可以利用多线程并行计算：桶1内的集合a和集合d的交集是x{3,5,7}，桶2内的集合b和集合e的交集是y{30, 50, 70}，桶3内的集合c和集合d的交集是z{}

最终，集合1和集合2的交集，是x与y与z的并集，即集合{3,5,7,30,50,70}

* + ***bitmap再次优化***

数据进行了水平分桶拆分之后，每个桶内的数据一定处于一个范围之内，如果集合符合这个特点，就可以使用bitmap来表示集合：

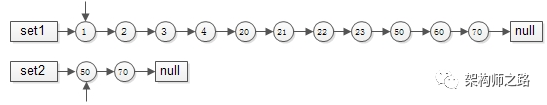


如上图，假设set1{1,3,5,7,8,9}和set2{2,3,4,5,6,7}的所有元素都在桶值[1, 16]的范围之内，可以用16个bit来描述这两个集合，原集合中的元素x，在这个16bitmap中的第x个bit为1，此时两个bitmap求交集，只需要将两个bitmap进行“与”操作，结果集bitmap的3，5，7位是1，表明原集合的交集为{3,5,7}

水平分桶，bitmap优化之后，能极大提高求交集的效率，但时间复杂度仍旧是O(n)。bitmap需要大量连续空间，占用内存较大

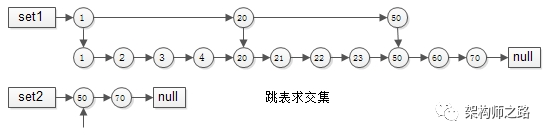
* + ***跳表skiplist***

有序链表集合求交集，跳表是最常用的数据结构，它可以将有序集合求交集的复杂度由O(n)降至O(log(n))



求集合1{1,2,3,4,20,21,22,23,50,60,70}和集合2{50,70}的交集，如果用拉链法，会发现1,2,3,4,20,21,22,23都要被无效遍历一次，每个元素都要被比对，时间复杂度为O(n)，能不能每次比对“跳过一些元素”呢？

跳表就出现了：



集合1{1,2,3,4,20,21,22,23,50,60,70}建立跳表时，一级只有{1,20,50}三个元素，二级与普通链表相同；集合2{50,70}由于元素较少，只建立了一级普通链表。

如此这般，在实施“拉链”求交集的过程中，set1的指针能够由1跳到20再跳到50，中间能够跳过很多元素，无需进行一一比对，跳表求交集的时间复杂度近似O(log(n))，这是搜索引擎中常见的算法。

* 1. ***总结***
* 全网搜索引擎系统由spider， search&index， rank三个子系统构成
* 站内搜索引擎与全网搜索引擎的差异在于，少了一个spider子系统
* spider和search&index系统是两个工程系统，rank系统的优化却需要长时间的调优和积累
* 正排索引（forward index）是由网页url\_id快速找到分词后网页内容list<item>的过程
* 倒排索引（inverted index）是由分词item快速寻找包含这个分词的网页list<url\_id>的过程
* 用户检索的过程，是先分词，再找到每个item对应的list<url\_id>，最后进行集合求交集的过程
* 有序集合求交集的方法有
  + 二重for循环法，时间复杂度O(n\*n)
  + 拉链法，时间复杂度O(n)
  + 水平分桶，多线程并行
  + bitmap，大大提高运算并行度，时间复杂度O(n)
  + 跳表，时间复杂度为O(log(n))

1. **流量从小到大，检索架构的演进**
   1. ***检索需求的满足与架构演进***

任何互联网需求，或多或少有检索需求，还是以58同城的帖子业务场景为例，帖子的标题，帖子的内容有很强的用户检索需求，在业务、流量、并发量逐步递增的各个阶段，应该如何实现检索需求呢？

* + 1. *原始阶段-LIKE*

数据在数据库中可能是这么存储的：t\_tiezi(tid, title, content)满足标题、内容的检索需求可以通过LIKE实现：select tid from t\_tiezi where content like ‘%天通苑%’,能够快速满足业务需求，存在的问题也显而易见：

* + 效率低，每次需要全表扫描，计算量大，并发高时cpu容易100%
  + 不支持分词
    1. *初级阶段-全文索引*

如何快速提高效率，支持分词，并对原有系统架构影响尽可能小呢，第一时间想到的是建立全文索引：alter table t\_tiezi add fulltext(title,content)，使用match和against实现索引字段上的查询需求。

全文索引能够快速实现业务上分词的需求，并且快速提升性能（分词后倒排，至少不要全表扫描了），但也存在一些问题：

* + 只适用于MyISAM
  + 由于全文索引利用的是数据库特性，搜索需求和普通CURD需求耦合在数据库中：检索需求并发大时，可能影响CURD的请求；CURD并发大时，检索会非常的慢；
  + 数据量达到百万级别，性能还是会显著降低，查询返回时间很长，业务难以接受
  + 比较难水平扩展
    1. *中级阶段-开源外置索引*

为了解决全文索引的局限性，当数据量增加到大几百万，千万级别时，就要考虑外置索引了。外置索引的核心思路是：索引数据与原始数据分离，前者满足搜索需求，后者满足CURD需求，通过一定的机制（双写，通知，定期重建）来保证数据的一致性。

原始数据可以继续使用Mysql来存储，外置索引如何实施？Solr，Lucene，ES都是常见的开源方案。推荐ES（ElasticSearch）而不是Lucene，原因是Lucene虽好，但始终有一些不足：

* + Lucene只是一个库，潜台词是，需要自己做服务，自己实现高可用/可扩展/负载均衡等复杂特性
  + Lucene只支持Java，如果要支持其他语言，还是得自己做服务
  + Lucene不友好，这是很致命的，非常复杂，使用者往往需要深入了解搜索的知识来理解它的工作原理，为了屏蔽其复杂性，一个办法是自己做服务

为了改善Lucene的各项不足，解决方案都是“封装一个接口友好的服务，屏蔽底层复杂性”，于是有了ES：

* + ES是一个以Lucene为内核来实现搜索功能，提供REStful接口的服务
  + ES能够支持很大数据量的信息存储，支持很高并发的搜索请求
  + ES支持集群，向使用者屏蔽高可用/可扩展/负载均衡等复杂特性

目前58到家使用ES作为核心，实现了自己的搜索服务平台，能够通过在平台上简单的配置，实现业务方的搜索需求。

ES完全能满足10亿数据量，5k吞吐量的常见搜索业务需求。

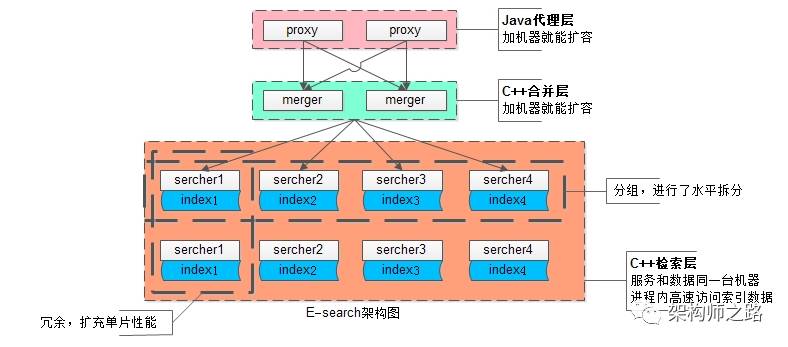
* + 1. *高级阶段-自研搜索引擎*

当数据量进一步增加，达到10亿、100亿数据量；并发量也进一步增加，达到每秒10万吞吐；业务个性也逐步增加的时候，就需要自研搜索引擎了，定制化实现搜索内核了。

* 1. **数据量、并发量、扩展性方案**

到了定制化自研搜索引擎的阶段，超大数据量、超高并发量为设计重点，为了达到“无限容量、无限并发”的需求，架构设计需要重点考虑“扩展性”，力争做到：增加机器就能扩容（数据量+并发量）。

以58同城的自研搜索引擎E-search为例，其初步架构图如下：

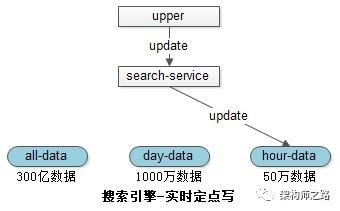


* + 上层proxy（粉色）是接入集群，为对外门户，接受搜索请求，其无状态性能够保证增加机器就能扩充proxy集群性能
  + 中层merger（浅蓝色）是逻辑集群，主要用于实现搜索合并，以及打分排序，业务相关的rank就在这一层实现，其无状态性也能够保证增加机器就能扩充merger集群性能
  + 底层searcher（暗红色大框）是检索集群，服务和索引数据部署在同一台机器上，服务启动时可以加载索引数据到内存，请求访问时从内存中load数据，访问速度很快
    - 为了满足数据容量的扩展性，索引数据进行了水平切分，增加切分份数，就能够无限扩展性能，如上图searcher分为了4组
    - 为了满足一份数据的性能扩展性，同一份数据进行了冗余，理论上做到增加机器就无限扩展性能，如上图每组searcher又冗余了2份

1. **搜索引擎的实时性**
   1. **索引分级**

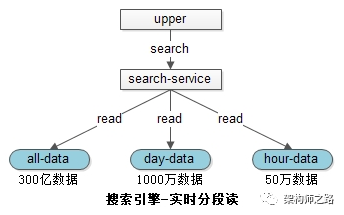
在数据量非常大的情况下，为了保证倒排索引的高效检索效率，任何对数据的更新，并不会实时修改索引，一旦产生碎片，会大大降低检索效率。

既然索引数据不能实时修改，可以通过将索引分为全量库、日增量库、小时增量库来保证最新的网页能够被索引到。



* 300亿数据在全量索引库中
* 1000万1天内修改过的数据在天库中
* 50万1小时内修改过的数据在小时库中

当有修改请求发生时，只会操作最低级别的索引，例如小时库。



当有查询请求发生时，会同时查询各个级别的索引，将结果合并，得到最新的数据：

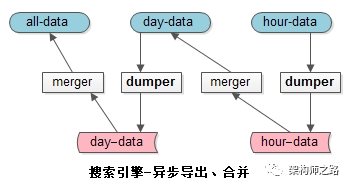
* 全量库是紧密存储的索引，无碎片，速度快
* 天库是紧密存储，速度快
* 小时库数据量小，速度也快

数据的写入和读取都是实时的，所以58同城能够检索到1秒钟之前发布的帖子，即使全量库有300亿的数据。

那么，小时库数据何时反映到天库中，天库中的数据何时反映到全量库中又成了新的问题。

* 1. **dump&merge**

这是由两个异步的工具完成的：



dumper：将在线的数据导出

merger：将离线的数据合并到高一级别的索引中去

小时库，一小时一次，合并到天库中去；天库，一天一次，合并到全量库中去；这样就保证了小时库和天库的数据量都不会特别大；如果数据量和并发量更大，还能增加星期库，月库来缓冲。

1. **长文本搜索去重\***
   1. 传统签名算法MD5与文本完整性判断
   2. 文本相似性的签名算法LSA
   3. minHash+simHash
   4. Jacccard相似性计算
   5. 再优化-分句

用标点符号把长文按照句子分开，使用N个句子集合（例如一篇文章中5条最长的句子作为签名，注意，长句子比短句子更具有区分性）作为文章的签名，在抄袭成风的互联网环境下，此法判断网页的重复度能大大降低工程复杂度，并且准确度也异常的高。

1. **短文本内存搜索\***

DAT是double array trie的缩写，是trie树的一个变体优化数据结构，它在保证trie树检索效率的前提下，能大大减少内存的使用，经常用来解决检索，信息过滤等问题。