第Ⅴ部分

案例学习

第 22 章

医疗公司塞纳(Cerner)的可聚合数据

(作者：Ryan Brush 与 Micah Whitacre)

一直以来，健康医疗信息技术所做的事情是将现有流程用自动化的方式来实现。然而这一切正在发生改变。随着人们对提高治疗质量和控制医疗成本的诉求日益增长，迫切需要有更好的系统来支撑这些目标。接下来，我们将看 Cerner 公司是如何用 Hadoop 生态系统来理解健康医疗的概念并构建方案来解决这些问题的。

22.1 从多 CPU 到语义集成

Cerner 公司长期致力于将技术应用于健康医疗，并在很长的一段时间里将重点放在电子医疗记录上。然而，新的问题需要一种更宽泛的方法，这驱使我们去研究 Hadoop。

2009 年时，我们需要为医疗记录建立更好的搜索索引。由此而引出的处理需求无法通过其他架构简单解决。搜索索引需要以高昂的代价处理临床文档：从文档中抽取术语，并解析其与其他术语的关系。例如，如果用户键入“心脏病”一词，我们希望返回的是探讨心肌梗塞的文档。这一处理，代价相当高，比如大文档，它需要占用数秒钟的 CPU 时间，而我们想要将其应用于数以百万计的文档。简而言之，我们需要投入许多 CPU 来处理这个问题，并且希望处理过程能够经济合算。

在其他可选方案中，我们曾经考虑过用一种基于分段式事件驱动架构(SEDA，staged

event-driven architecture)的方法来成规模地抽取文档。但是 Hadoop 在满足一项重要需求方面脱颖而出：我们需要在数小时或者更快的时间内频繁地反复处理数以百万计的文档。面向临床文档的知识抽取的逻辑在快速改进中，我们需要将这种改进迅速推向世界。在 Hadoop 中，这仅仅意味着在已有的数据之上运行一个新版本的 MapReduce 作业，然后处理文档被载入一个 Apache Solr 服务器集群以支持应用查询。

这些早期的成功为后期更多涉足的项目打好了基础。这一系统类型及其数据可被用作经验基础，以帮助控制成本以及改善全部人口的医疗水平。由于健康医疗数据经常以碎片化的形式分布在各系统和机构中，我们需要首先收集所有这些数据并理解其含义。

当有了大量的数据源和格式，甚至标准化的数据模型有待解析时，我们面临着一个庞大的语义集成问题。最大的挑战并非来自数据的规模——众所周知 Hadoop 可以根据需要扩展——而是来自于为满足我们的需求，对数据进行清理、管理和转换所带来的极端复杂性。我们需要更高级的工具来管理这一复杂性。

22.2 进入 Apache Crunch

收集并分析这些各不相干的数据集会引发出许多需求，其中以下几点最为突出。

- 需要将许多处理步骤分割成模块，这些模块可被轻易地组装进一条复杂的管线。
- 需要提供一个比原始 MapReduce 更高级的编程模型。
- 需要处理医疗记录的复杂结构，该结构具有数百个独一无二的字段和数级嵌套子结构。

在这一案例中，我们尝试过多种选择，包括 Pig、Hive 和 Cascading，每一种效果都不错。我们继续使用 Hive 进行一些特别的分析处理，然而当把任意逻辑应用于我们的复杂数据结构时，Hive 却变得很不灵便。然后我们听说了 Crunch(参见第 18 章)，Josh Wills[①]领导的一个项目，与 Google 的 FlumeJava 系统相类似。Crunch 提供了一个简单的基于 Java 的编程模型以及对记录的静态类型检查，完美贴合我们 Java 开发者社区的需求以及所处理的数据类型。

① 编注：前谷歌广告系统工程师，曾经专门写书讲述了工业界和学术界在机器学习方面的异同，现为著名信息聚合平台 Slack 的首席数据工程师。

22.3 建立全貌

规模化的理解和管理健康医疗需要大量干净、规范化和可靠的数据。不幸的是，这样的数据通常分布在许多数据源中，合并起来较为困难且易于出错。医院、医生办公室、诊所和药房各自拥有个人记录的一部分，以行业标准的格式，诸如CCDs(Continuity of Care Documents)、HL7(Health Level 7，一种健康医疗数据交换格式)、CSV 文件或专有格式存储。

我们面临的挑战是，如何获取这一数据，将其转换为一种干净、集成的表现形式，并使用它创建注册信息，以帮助患者管理明确的病情、度量健康医疗的可操作方面，并支持各种分析，如图 22-1 所示。

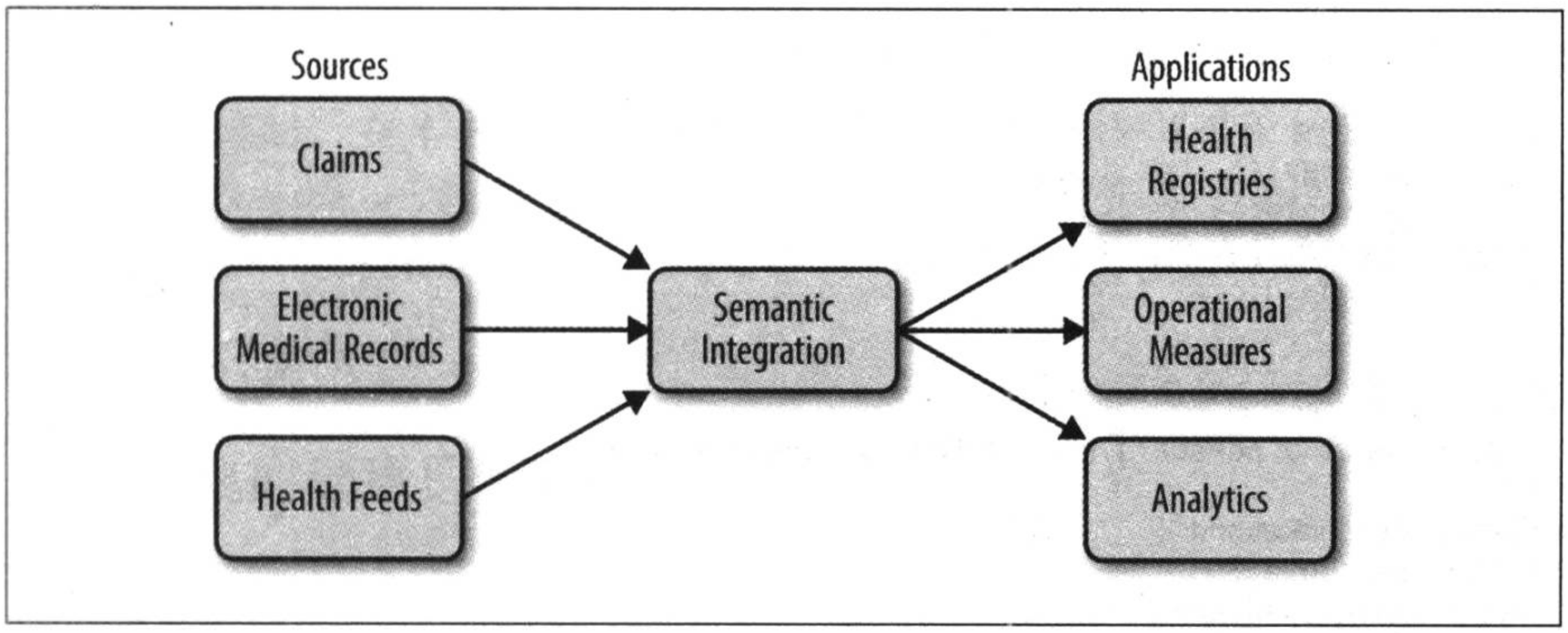

图 22-1. 可操作的数据流

一个至关重要的步骤是，创建一个我们可以依赖的干净的、语义集成的基础，这也是本章实例学习的聚焦点。我们先将数据规范化为一种通用结构。这一系统的早期版本使用了不同的模型，但是后来都已经迁移至 Avro 以便在各处理步骤之间存储和共享数据。范例 22-1 给出了一个简化的 Avro IDL 以描述我们提出的通用结构。

范例 22-1. 用于通用数据类型的 Avro IDL

```
@namespace("com.cerner.example")
protocol PersonProtocol {

  record Demographics {
  string firstName;
  string lastName;
  string dob;
  ...
```

```
}
record LabResult {
string personId;
string labDate;
int labId;
int labTypeId;
int value;
}
record Medication {
string personId;
string medicationId;
string dose;
string doseUnits;
string frequency;
...
}
record Diagnosis {
string personId;
string diagnosisId;
string date;
...
}
record Allergy {
string personId;
int allergyId;
int substanceId;
...
}
/**
* Represents a person's record from a single source.
*/
  record PersonRecord {
  string personId;
  Demographics demographics;
  array<LabResult> labResults;
  array<Allergy> allergies;
  array<Medication> medications;
  array<Diagnosis> diagnoses;
  . . .
  }
}
```

注意，这里多种数据类型都嵌套在一条通用个人记录中，而不是分散在不同数据集中。这种结构支持对这类数据的最常见的使用模式(查看一条完整的记录)无需对数据集进行大量昂贵的连接操作。

将数据写入 PCollection<PersonRecord>使用了一系列 Crunch 管线，这样可以隐藏每个源的复杂性，并提供了一个与原始规范化记录数据进行交互的简单接口。在幕后，每个 PersonRecord 可以存储在 HDFS 中或者作为 HBase 的一行存储，其中每个人的数据元素分布在列簇(column families)和列描述(column qualifiers)中。聚合的结果看起来如表 22-1 所示。

表 22-1. 聚合数据

源	个人 ID	个人统计信息	数据
医生办公室	12345	Abraham Lincoln…	糖尿病诊断，实验室结果
医院	98765	Abe Lincoln ...	流感诊断
药房	98765	Abe Lincoln ...	过敏，药物
诊所	76543	A. Lincoln ...	实验室结果

希望从一些获得授权的源中检索数据的用户调用一个“retriever” API，就可以轻松地为被请求数据生成一个 Crunch `PCollection`：

```
Set<String> sources = ...;
PCollection<PersonRecord> personRecords =
   RecordRetriever.getData(pipeline, sources);
```

这种 retriever 模式允许用户载入数据集，而无需知道这些数据集物理上是如何及在哪里存储的。在本书编写的同时，这一模式的某些用途正被新兴的 Kite SDK(*http://kitesdk.org/*)所取代，用于在 Hadoop 中管理数据。检索到的`PCollection<PersonRecord>`中的每个条目代表了一个人在单一源中的完整医疗记录。

22.4 集成健康医疗数据

从原始数据到健康医疗相关的问题答案之间，有着许多处理步骤。这里我们讨论其中一个步骤：将来自多个源的、关于同一个人的数据集中起来。

不幸的是，在美国缺乏通用的病人标识符，再加上噪声数据的影响，例如不同系统中个人姓名和统计资料的变异，使得跨越多个源将一个人的数据精确地统一起来非常困难。分布在多个源中的信息可能看起来如表 22-2 所示。

表 22-2. 来自多个源的数据

源	个人ID	名	姓	地址	性别
医生办公室	12345	Abraham	Lincoln	1600 Pennsylvania Ave.	M
医院	98765	Abe	Lincoln	Washington, DC	M
医院	45678	Mary Todd	Lincoln	1600 Pennsylvania Ave.	F
诊所	76543	A.	Lincoln	Springfield, IL	M

在健康医疗领域，上述数据统一问题通常是通过一个称为“企业级患者主索引”(EMPI，Enterprise Master Patient Index)的系统来解决的。一个 EMPI 可以从多个系

统中获取数据，并确定哪些记录确实是对应了同一个人。有很多途径可以达到这种目的，例如，明确声明各自关系的人类，或是能够识别共性的复杂算法。

在某些情况下，我们可以从外部系统载入 EMPI 信息，而在另一些情况下，我们可以在 Hadoop 内对其进行计算。关键在于我们可以将这一信息提供给基于 Crunch 的管线使用。其结果便是 PCollection<EMPIRecord>，数据结构如下：

```
@namespace("com.cerner.example")
protocol EMPIProtocol {

  record PersonRecordId {
  string sourceId;
  string personId
  }
  /**
  * Represents an EMPI match.
  */
  record EMPIRecord {
  string empiId;
  array<PersonRecordId> personIds;
  }
}
```

给定这一结构中数据的 EMPI 信息，PCollection<EMPIRecord>将会包含类似于表 22-3 所示的数据。

表 22-3. EMPI 数据

EMPI 标识符	个人记录 ID(<源 ID，个人 ID>)
EMPI-1	<offc-135, 12345>
	<hspt-246, 98765>
	<clnc-791, 76543>
EMPI-2	<hspt-802, 45678>

为了将所提供的 PCollection<EMPIRecord>和 PCollection<PersonRecord>里属于同一个人的医疗记录组合起来，必须将所收集的数据转换成 PTable，并采用一个通用键做为键。在这种情况下，Pair<String,String>(其中第一个值为 sourceId，第二个值为 personId)将确保采用唯一的键值进行连接。

第一步是从所收集的数据中的每个 EMPIRecord 里提取通用键：

```
PCollection<EMPIRecord> empiRecords = ...;
PTable<Pair<String, String>, EMPIRecord> keyedEmpiRecords =
    empiRecords.parallelDo(
  new DoFn<EMPIRecord, Pair<Pair<String, String>, EMPIRecord>>() {
```

```
      @Override
      public void process(EMPIRecord input,
        Emitter<Pair<Pair<String, String>, EMPIRecord>> emitter) {
      for (PersonRecordId recordId: input.getPersonIds()) {
      emitter.emit(Pair.of(
          Pair.of(recordId.getSourceId(), recordId.getPersonId()), input));
        }
      }
    }, tableOf(pairs(strings(), strings()), records(EMPIRecord.class)
  );
```

下一步，需要从每个 PersonRecord 里提取相同的键：

```
PCollection<PersonRecord> personRecords = ...;
PTable<Pair<String, String>, PersonRecord> keyedPersonRecords =
personRecords.by(
    new MapFn<PersonRecord, Pair<String, String>>() {
  @Override
  public Pair<String, String> map(PersonRecord input) {
    return Pair.of(input.getSourceId(), input.getPersonId());
  }
}, pairs(strings(), strings()));
```

连接这两个 PTable 对象将返回 PTable<Pair<String, String>,Pair<EMPIRecord, PersonRecord>>。此时，原键不再有用，于是我们用 EMPI 标识符作为该表的键：

```
PTable<String, PersonRecord> personRecordKeyedByEMPI = keyedPersonRecords
    .join(keyedEmpiRecords)
    .values()
    .by(new MapFn<Pair<PersonRecord, EMPIRecord>>() {
  @Override
  public String map(Pair<PersonRecord, EMPIRecord> input) {
    return input.second().getEmpiId();
  }
}, strings()));
```

最后一步是通过键将表进行分组，以保证所有要作为一个完整集合处理的数据被聚合在一起：

```
PGroupedTable<String, PersonRecord> groupedPersonRecords =
  personRecordKeyedByEMPI.groupByKey();
```

PGroupedTable 将包含类似表 22-4 中的数据。

这种将数据源进行统一的逻辑只是一个更大规模执行流程的第一步。其他一些 Crunch 下游功能基于这些步骤构建，以满足众多客户需求。在一个常见应用案例中，通过将统一的 PersonRecord 的内容载入一个基于规则的处理模型以发布新的临床诊断知识，这种方式解决了大量问题。例如，我们可以在那些记录之上运

行规则，从而判断一个糖尿病患者是否正在接受推荐疗法，并指出可以改进之处。相似的规则集为了各种需求而存在，从一般健康问题到复杂病情管理。这种逻辑可以是复杂的，并且对于不同的应用案例会存在许多变种，但逻辑的实现全都依赖于构成 Crunch 管线的各功能模块。

表 22-4. 已组合的 EMPI 数据

EMPI 标识符	可迭代的个人记录
EMPI-1	{ "personId": "12345", "demographics": { "firstName": "Abraham", "lastName": "Lincoln", ... }, "labResults": [...] }, { "personId": "98765", "demographics": { "firstName": "Abe", "lastName": "Lincoln", ... }, "diagnoses": [...] }, { "personId": "98765", "demographics": { "firstName": "Abe", "lastName": "Lincoln", ... }, "medications": [...]}, { "personId": "76543", "demographics": { "firstName": "A.", "lastName": "Lincoln", ... } ... }
EMPI-2	{ "personId": "45678", "demographics": { "firstName": "Mary Todd", "lastName": "Lincoln", ... } ... }

22.5 框架之上的可组合性

这里所描述的模式，呈现了以个人为中心的健康医疗的一个特定类别的问题。然而，这一数据也可用作理解健康医疗的可操作性和系统性的基础，这对我们转换

和分析它的能力提出了新的要求。

像 Crunch 这样的库能够帮助我们应对新兴的需求，因为它们有助于将我们的数据和处理逻辑变得可组合化。相比单一、静态的数据处理框架，我们能够将功能和数据集模块化，并且在新需求出现时对其进行重用。图 22-2 展示了组件如何以新的方式彼此连接，其中每个方框作为一个或多个 Crunch DoFns 来实现。这里，我们利用个人记录来识别糖尿病患者，推荐健康管理计划，同时采用那些可组合部件来整合可操作的数据并推动健康系统分析。

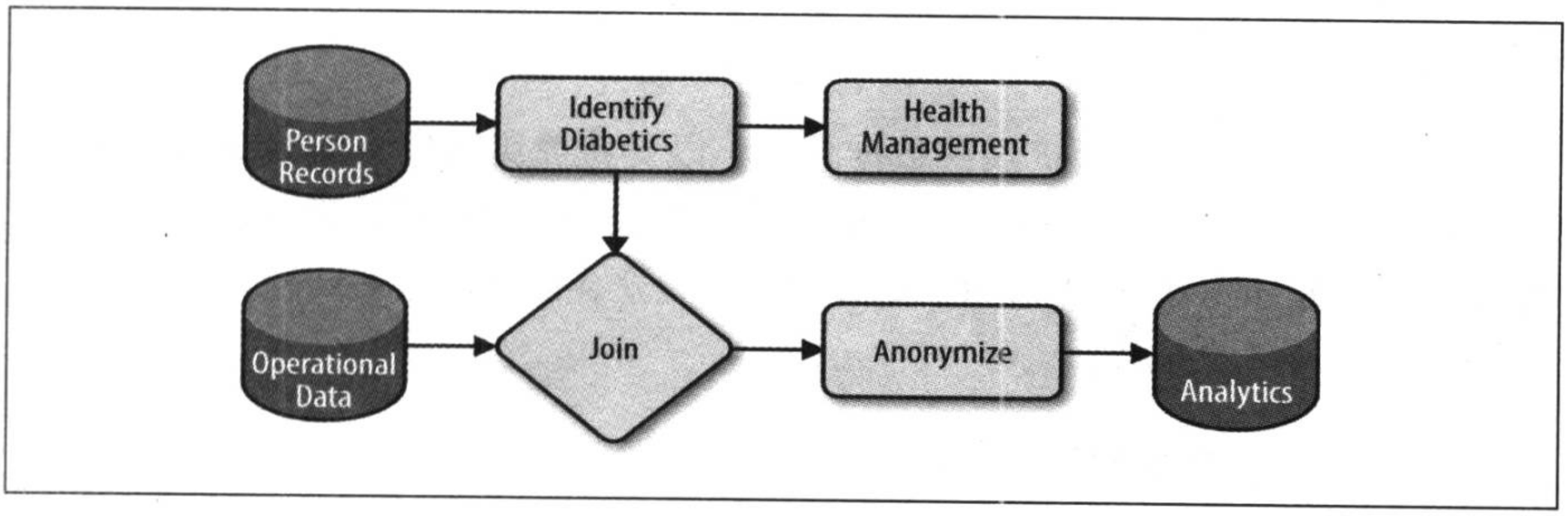

图 22-2. 可组合的数据集和功能

可组合性也使得对新问题空间的迭代变得更简单。当创建一种新的数据视图以回答一类新问题时，我们能够借助现有的数据集和转换过程，发布我们的新版本。随着问题变得更易于理解，该视图可以被迭代替换或更新。最终，这些新功能和数据集对历史资产做出贡献，并被用于新的需求。其结果便是一个不断增长的数据集目录，以支持不断增长的理解数据的需求。

处理过程需要与 Oozie 相互协调。每当新的数据到达，在 HDFS 中的一个明确定义的位置，将创建一个具有唯一标识符的新数据集。Oozie 协调器(Oozie coordinators)观察该位置，并简单地发起 Crunch 任务来创建下游数据集，该数据集随后可被其他协调器获得。在本书写作时，数据集和更新由 UUIDs 标识以保持其唯一性。然而，我们正在进行一项处理，即将新数据放置在基于时间戳的分区(timestamp-based partitions)中，以便更好地与 Oozie 的标称时间模型(Oozie's nominal time model)一起工作。

22.6 下一步

我们正在关注两个主要步骤，从而更高效地将这一系统的价值最大化。

首先，我们想要创建关于 Hadoop 生态系统及其支持库的规范化实践。本书和其他地方都定义了很多优秀的实践，但是经常需要重要的专业知识来有效实现这些实践。我们正在使用和建立库，从而使此类模式对于更大规模的受众来说变得明晰且易得。

通过将各种连接和处理模式构建进库，Crunch 提供了一些好的例子。

其次，我们不断增长的数据集目录引发了对简单且规范的数据管理的需求，这也是对 Crunch 提供的处理特性的一个补充。在一些使用案例中，我们已采纳 Kite SDK 来满足这一需求，并希望有朝一日扩展其用途。

最终目标为一个安全、可扩展的数据目录，以支持健康医疗领域的众多需求，包括尚未出现的问题。事实证明 Hadoop 能够满足我们对数据和处理的需求，而更高级的库正在使 Hadoop 广泛适用于更大规模的受众，以解决诸多问题。