

□ 前言

- 为什么需要“大规模计算” [HPC, DL, Business platform system, Cloud已经合流]
 - 导入 – 科学计算(天气预报), DL, 互联网平台(Google, Amazon, Alibaba, MeiTuan, ...)

□ 基础篇

- 并发程序的样子 – Divide & Conquer, Model & Challenges, PCAM, Data/Task, ...
 - 天气预报的计算
- 运行环境
 - 硬件 – 自己梳理的3个方案 – Shared/Unshared Memory, Hybrid
 - 系统软件 – 协议栈, Modern OS, Distributed Job Scheduler, GTM等

□ 算法级篇

- OpenMP, MPI, CUDA (DL的实现), Big Data 中的MR/Spark等 (只涉及在Big Data SDK之上的编程; 大数据本身的介绍放到后一部分)

□ 系统级篇 – 互联网平台的实现

- “秒杀”的技术架构
- 计算广告
- 系统架构 (HTAP等)
 - Flink, ClickHouse, MaxCompute, ELK ...

Chapter 4: Distributed OS

□ Support the execution of many execution units – parallel or distributed

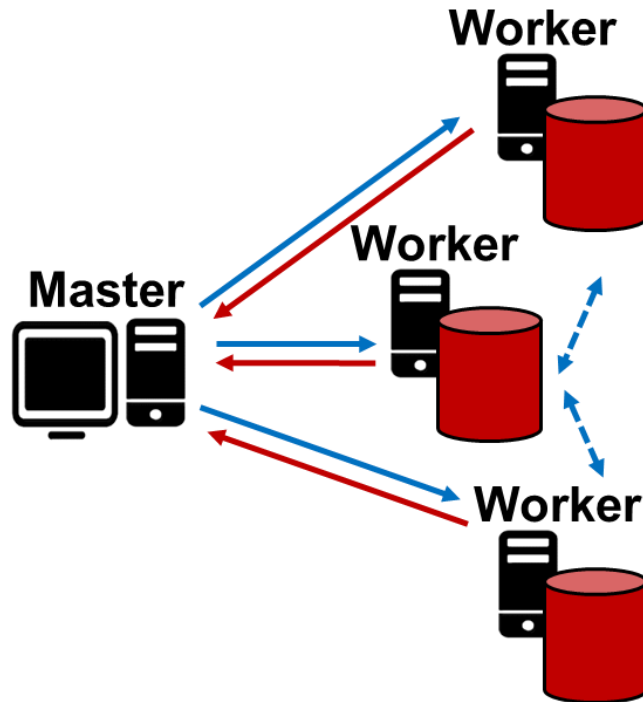
- OS's Primary function is to support the execution of many (distributed) programs - **Protocols**
 - **Resource availability & Dispatching**
 - Successful cooperation – circumstance is stable or not
- Evolution of related frameworks/platforms
 - From Amoeba [变形虫] to Micro-services [微服务]
- Other large scale computing systems

Parallel program on distributed system (with many computers)

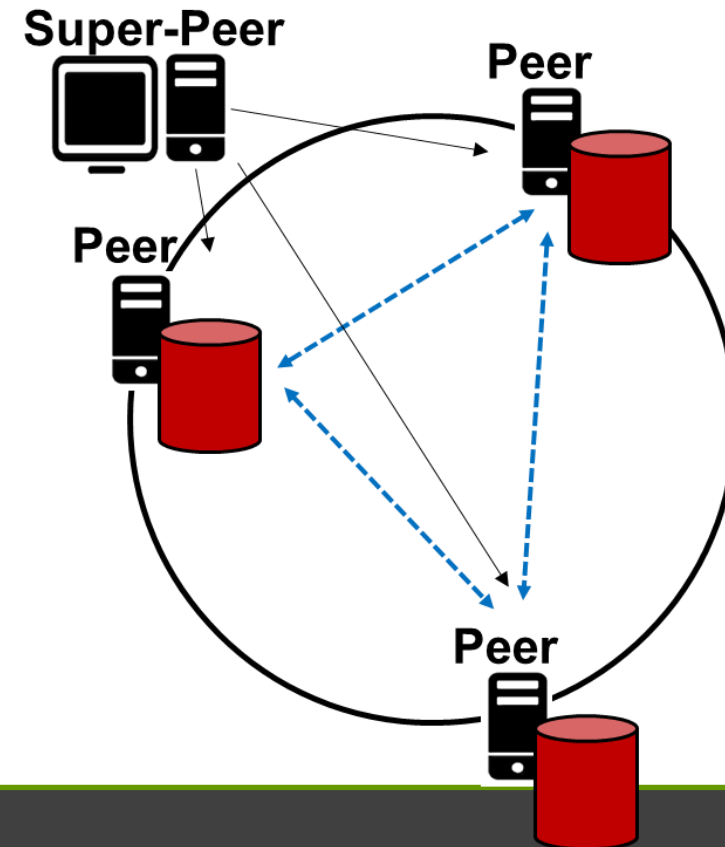
□ (like **Cluster**)

■ 2 main architectures in distributed computing

Master-Slave

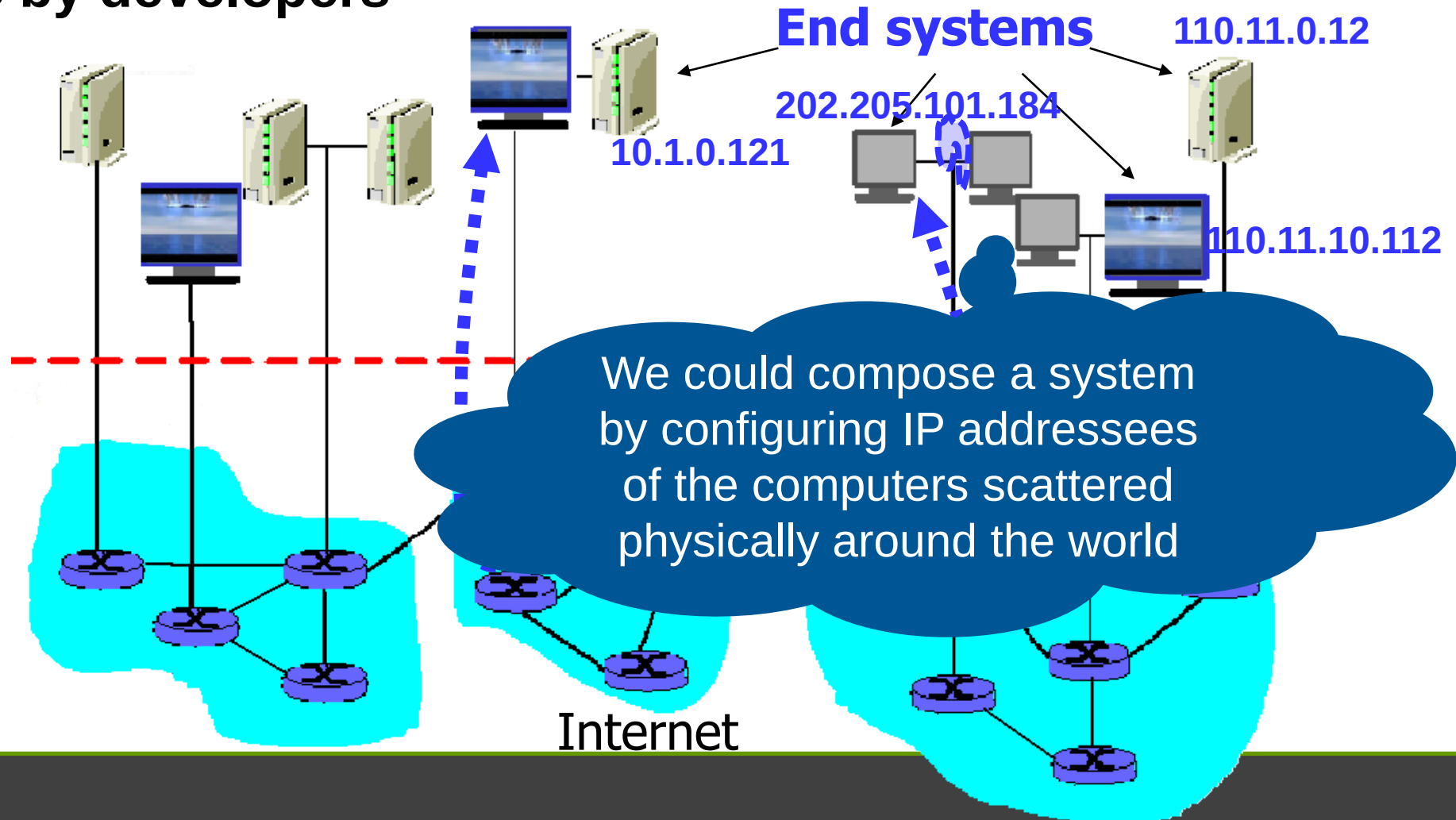


Peer-to-Peer



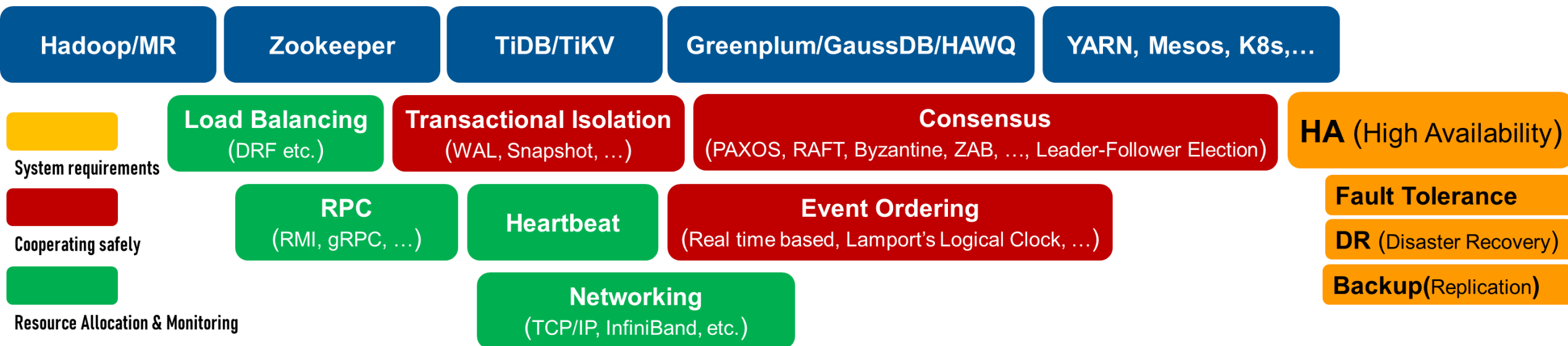
Overlay network

- For PCs as computers or containers, they are usually configured in advance by developers



□ There are many other systems following similar ideas

- DNS, P2P
- Storage systems, Cache,
- Business systems (discussed in later chapter)
- ...



分布式数据管理 – 之我的理解

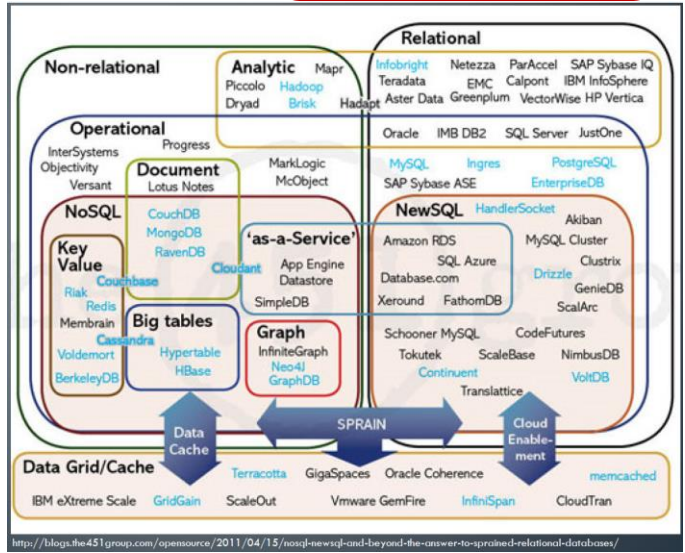
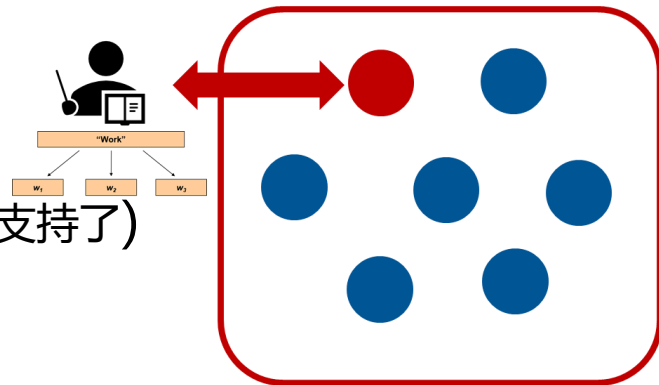
□ 沿用前面关于分布式软件的通用结构，分布式数据管理系统(应该涵盖 NoSQL, NewSQL, 以及传统数据库管理系统的分布式化)既要满足数据管理的规则，也要体现分布式的特点

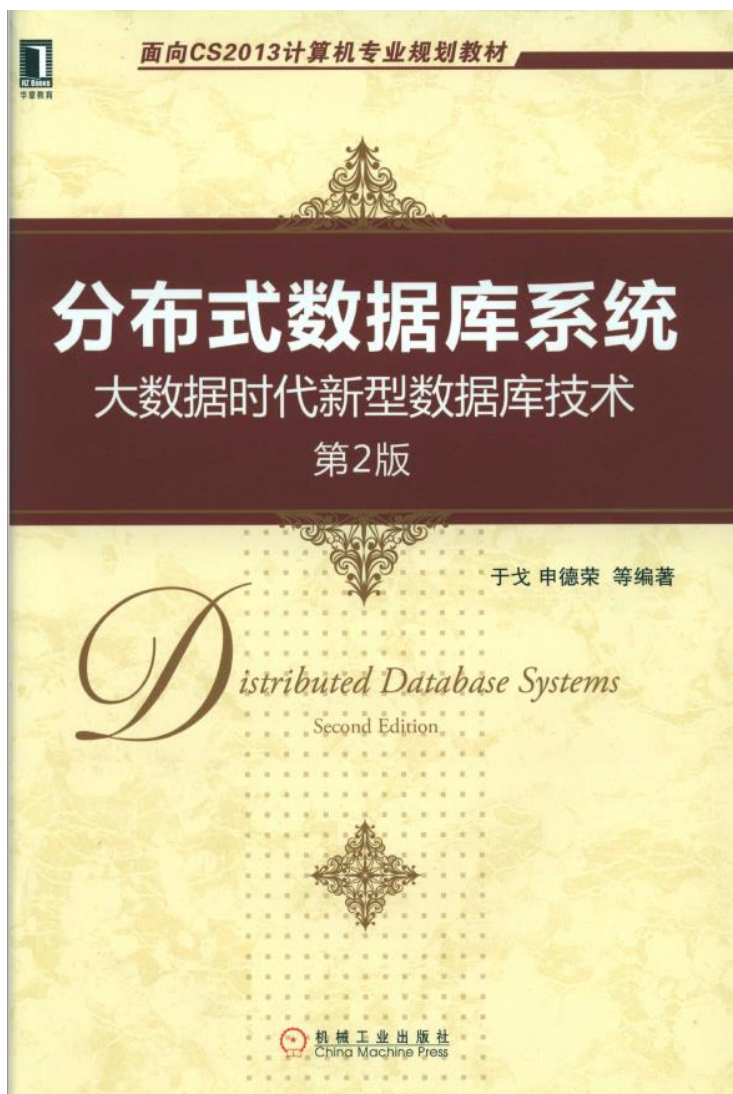
□ 数据管理

- 多用户并发访问
- SQL仍然发挥作用(虽然有NoSQL的过渡，但在NewSQL又把NoSQL纳入SQL的支持了)
- 数据的一致性

□ 分布式的特点

- 海量数据存取 → 那么数据必然是**分布式存储**，也就意味着**SQL需要分布式执行**
- 容错 (Fault Tolerance) → 保障数据安全，思路也就是**冗余**而已；并且记录数据处理的动态
- HA (High Availability) → 服务节点也**冗余**嘛 (自然要保证冗余的服务节点与现活服务节点保持状态一致)
- . . .





- ❑ 分布式数据库系统：大数据时代新型数据库技术（第2版）
- ❑ 于戈 申德荣等
- ❑ **2021**
- ❑ 机械工业出版社

Distributed DBMS – since Mid of 1970s

数据库系统的发展起始于 20 世纪 60 年代，从 IBM 的层次模型 IMS、网状模型、关系模型，发展到多数据模型共存。随着科学技术的发展，各个行业、领域对数据库技术提出了更多的需求，推动了数据库技术同诸多新技术如分布式处理技术、并行计算技术、人工智能技术、多媒体技术、模糊计算技术等相结合，由此衍生出了多种新的数据库技术。分布式数据库系统是其中的一种新数据库技术。分布式数据库系统兴起于 20 世纪 70 年代中期。推动分布式数据库系统发展的动力来自于两方面：一是应用需求，二是硬件环境的发展。在应用需求上，全国甚至全球范围内的航空及铁路订票系统、银行通存通兑系统、水陆空联运系统、跨国公司管理系统、连锁配送管理系统等，都涉及地理上分布的企业或机构的局部业务管理和与整个系统有关的全局管理，采用传统的集中式数据库管理系统已无法满足这种分布式应用需求。在硬件环境上，提供了功能强大的计算机和成熟的广域公用数据网及快速增长的局域网。在上述两方面的推动下，人们期望符合现实需要的、能处理分散地域的、具备数据库系统特点的新数据库系统的出现。

从 20 世纪 70 年代中期开始，各发达国家纷纷投巨资支持分布式数据库系统的研究和开发计划。历时十年，呈现出了许多研究成果。典型的原型系统有美国国防部委托 CCA 公司设计和研制的 SDD-1 分布式数据库系统、美国加利福尼亚大学伯克利分校研制的 分布式 INGRES 系统、IBM 圣何塞实验室研制的 R* 分布式数据库系统、德国斯图加特大学研制的 Porel 分布式数据库系统、法国 Sirius 资助计划产生的若干原型系统（如 Sirius-Delta、Polypheme 等）。随后，商品化的数据库系统 Oracle、Sybase、DB2、Informix、INGRES 等都从分布式数据库系统研究中吸取了许多重要的概念、方法和技术，实现了相当程度上的分布式数据管理功能，并宣称它们都是分布式数据库系统产品。在分布式数据库系统的商品化进程中，随着研究的深入和应用的普及，更由于分布式数据库管理系统本身的高复杂性，研究者提出了更简洁、更灵活的实现技术来满足分布式数据处理的要求。目前，商品化数据库产品如 Oracle、Sybase、DB2、SQL Server、Informix 都支持异构数据库系统的访问和集成功能。它们都采用基于组件和中间件的松散耦合型事务管理机制来实现分布式数据的管理，具有高灵活性和可扩展性，并且具有替代传统分布式数据库管理系统中的紧耦合型事务管理机制的趋势。

随着技术的发展，大数据广泛存在，如 Web 数据、移动数据、社交网络数据、电子商务数据、企业数据、科学数据等，并且各行各业都期望得益于大数据中蕴含的有价值的知识。为此，呈现出了支持大数据管理和分析的技术，如大数据存储模型、键值模型、MapReduce 分布式处理架构、改进的支持分布式的事务协议、副本管理等，并推出了许多关系云系统和多存储结构的大数据库系统等。支持大数据库管理的基础理论和技术，典型代表是以经典的分布式数据库理论和技术为基础的扩展研究，满足大数据处理的实时性、高性能和可扩展性需求等。

多年来，作者在国家自然科学基金、国家 973 计划、国家 863 计划等课题的支持下，以大数据管理、Web 数据库集成、联盟企业数据集成应用背景，针对分布式环境下的数据管理进行了深入研究。同时，作者一直承担东北大学计算机专业硕士研究生的分布式数据库系统课程以及计算机专业本科生的数据库系统概论和数据库系统实现课程的教学工作。本书正是基于以上工作而撰写的。



Oracle的分布式在中国

2.9.3 Oracle 分布式数据库架构

OraStar 是一家跨国公司，总部设在中国北京。为了获得较低的人力成本和方便的配件物流，OraStar 将产品的生产工厂设置在广东，并在广州设立了对应的生产部门。为了更好地掌握营销渠道，OraStar 将销售部门安排在上海。同时，OraStar 还在美国的旧金山建立了海外总部，在日本东京设立了研发中心。

OraStar 利用 Oracle 分布式数据库架构来管理该公司的信息数据。在公司的总部，生产部门和销售部门分别采用 Oracle 10g 管理本部门涉及的信息数据。各部门之间的 Oracle 通过数据库链 DBLink 相互连接。利用 DBLink，本地的数据库用户可以访问远程数据库中的数据，例如，总部的用户可以查询生产部门数据库中的数据，也可以查询销售部门数据库中的数据，反之亦然。通过设计，这种访问对用户来说是透明的，即用户不必了解数据的具体存放地点。

利用高级复制技术，OraStar 公司将北京总部的数据同步到美国旧金山海外总部的 Oracle 11g 数据库中，这样可以减少网络流量，提高这部分数据的访问速度和可用性。而位于上海的销售部门利用 Oracle Streams 技术，将数据复制到东京的研发中心 Oracle 12c 数据库中。研发中心整理、分析、挖掘这部分数据，为公司的营销决策提供有效的预测。同时研发中心有海量的测试数据保存在 Oracle 大数据机中。Oracle 大数据机是一个功能完备的大数据平台，集成了 Hadoop、Oracle NoSQL Database、Enterprise Linux 等模块，旨在以较低的总拥有成本进行安全可靠的数据处理。研发中心可以利用 Hadoop 分布式文件系统直接连接器访问存储在 Hadoop 平台上的数据。

为 OraStar 公司的生产部门提供配件的供货商使用的是 SQL Server 数据库。OraStar 利用 Oracle 提供的异构服务，访问远程的 SQL Server 数据库，及时了解配件的库存信息。

综上所述，图 2-20 描绘了一个比较完整的基于 Oracle 的分布式数据库应用案例。位于不同场地内的每个 Oracle 数据库可以实现节点自治。节点之间通过 DBLink 相互连接，通过适当的设计(将在第 3 章中介绍)，可以实现用户对数据所在的场地透明。同时，利用高级复制和 Oracle Streams 技术将数据的副本同步到远程的同构数据库中。对于非 Oracle 的数据库节点，利用异构服务组件使之与 Oracle 数据库相连，组成一个异构的分布式数据库架构。

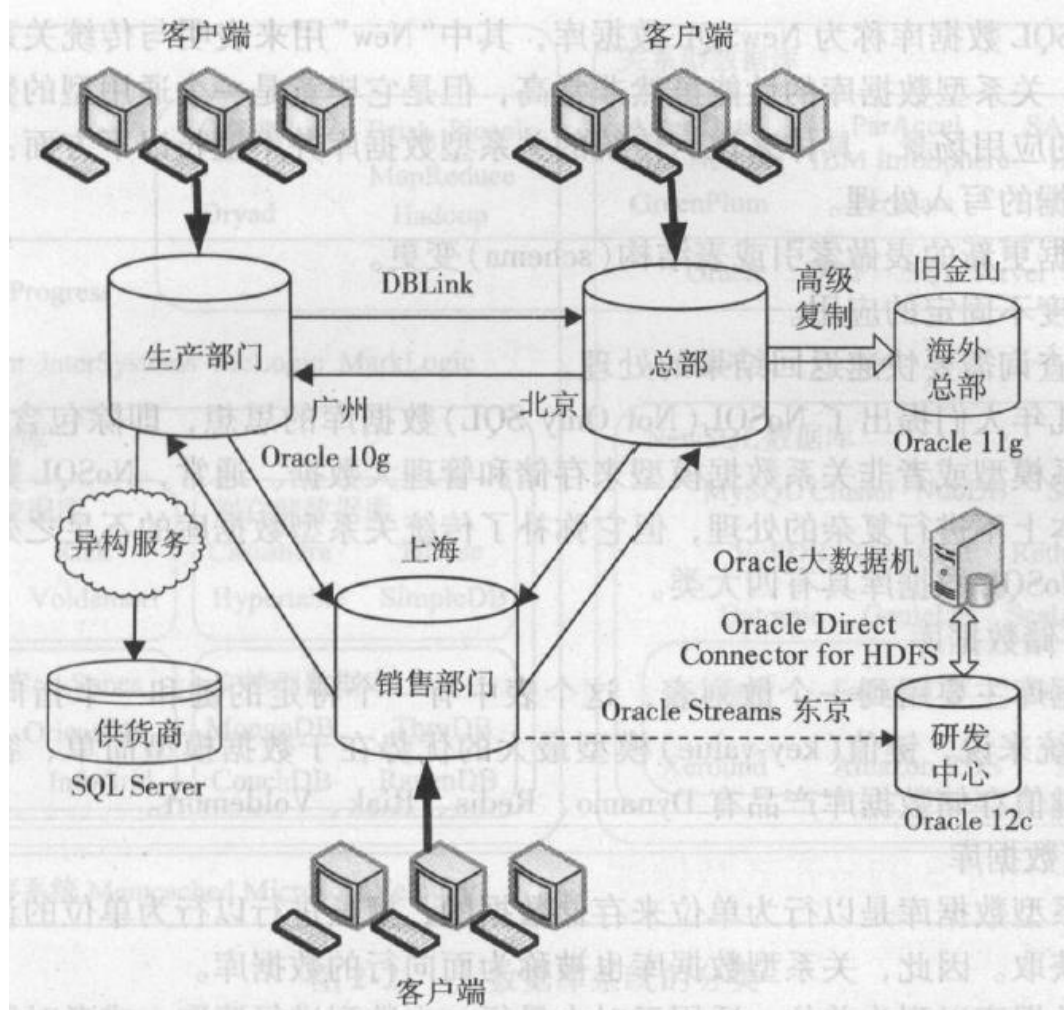


图 2-20 Oracle 分布式数据库架构

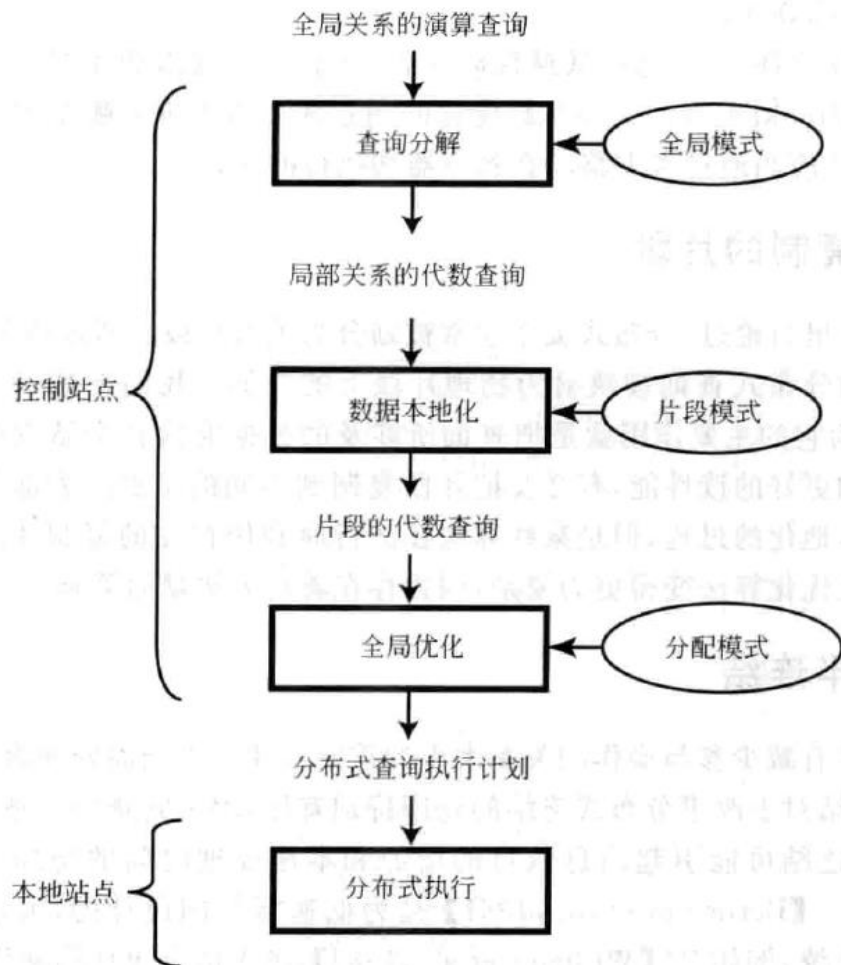


图 6.3 分布式查询处理的通用分层结构

6.5.1 查询分解

第一层把演算查询分解为全局关系上的代数查询，这一转换所需的信息可以在描述全局关系的全局概念模式里找到。但是这里不需要数据分布的信息，这一信息用于下一层。所以这一层所需的技术属于集中式 DBMS 的范围。

查询分解由四个连续的步骤组成。

首先，演算查询重写成规范的形式以适合后面的处理。查询的规范化一般要涉及查询量词的操作，以及应用逻辑操作优先级来验证查询是否合格。

第二，在语义上对规范化的查询分析，从而能够尽早地检查和拒绝不正确的查询。现有的技术仅仅能够用于关系演算的一个子集，通常这些技术都使用某种图来表达查询的语义。

第三，对正确的查询（仍然是关系演算的形式）进行简化，简化的方法之一就是去掉重复的谓词。注意，当查询是由系统对用户查询实行转换而生成的结果时，很可能产生重复的查询。我们会在第 5 章里看到，这些转换要用于完成语义数据的控制（视图，保护，以及予以完整性控制）。

第四，演算查询重构为一个代数查询。我们在 6.1 节讨论过，从同一个演算查询可以推导出多个等价的代数查询，而其中的某些代数查询会比另一些“更好”。一个代数查询的质量要用期待的性能来衡量，传统的通过转换获得“更好”的代数表示的方法是从一个

6.5.2 数据本地化

第二层的输入是针对全局关系的一个代数查询，这一层的主要工作就是利用片段模式的数据分布信息对查询的数据实行本地化。在第 3 章里我们看到了关系被分成了互不相交的、称为片段的子集，每个子集存放在不同的站点上。这一层要判断哪些片段被用到查询中，并且把分布查询转换为在片段上的查询。通过关系运算我们能够表达分片谓词，实现片段的定义。通过应用分片规则可以对一个全局关系进行重构，然后推导出一个由关系代数运算构成的本地化程序（localization program），由该程序作用在片段上。在片段上生成查询由两步来完成。首先，通过把查询的关系替换成重构计划（也称为物化计划（materialization plan））实现从查询到片段查询的映射，这里提到的重构计划已在第 3 章讨论过。第二，对片段查询进行简化和重构，生成新的“好的”查询。简化和重构可以按照查询分解层所采用的原理进行。如同查询分解层那样，最后生成的片段查询一般都与最优查询有着相当的距离，因为有关片段的信息还没有得到利用。

6.5.3 全局查询优化

第三层的输入是针对片段的代数查询。查询优化的目的执行策略，要明白寻找最优的结果在计算上是不可行的。分布式代数运算，以及在站点间传输数据的通信元语（communication operation）加以描述。上一层已经优化了查询，例如去掉了重复的特征，例如片段的分配和基数等无关。此外，也没有包括通的一个查询内的操作顺序的排列，可以发现许多等价的查询。

查询优化由发现查询操作的“最佳”排序所组成，其中包括代价函数一般用时间加以定义，要参考诸如磁盘空间、磁盘 I/O 等方面的计算资源。早期分布式 DBMS 所作的一个典型的假设是主要的因素，我们在前面已经指出过这一点。这一假设在宽带网而言，其有限的带宽使得通信成本比本地处理更为昂贵。的成本可能会比 I/O 的成本还要小。为了选择操作的排序，不执行代价。在执行前（例如静态优化）决定执行代价要以片段的公式为基础。所以，优化决策要依赖于片段的分配以及可用数据都记录在片段分配的方案里。

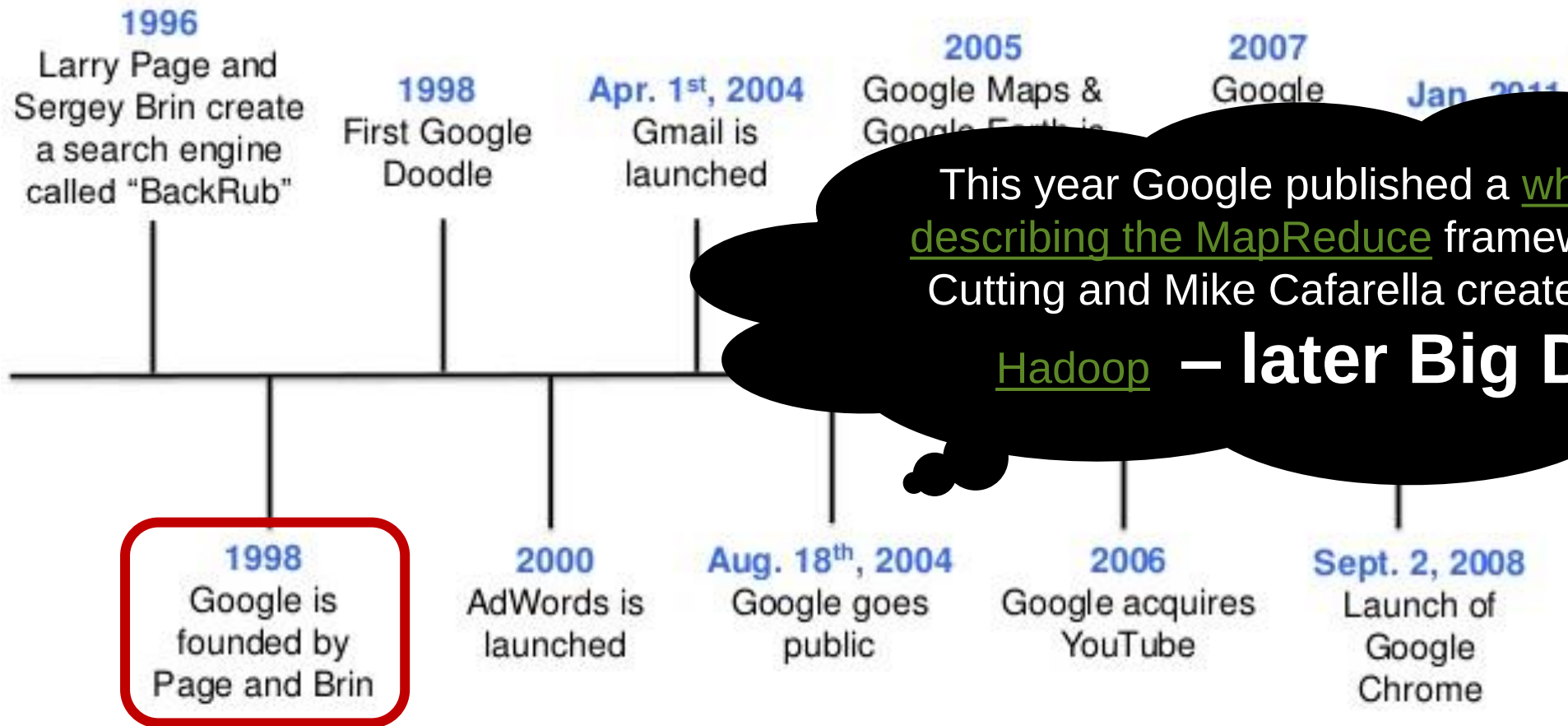
查询优化的一个重要的方面是连结顺序（join ordering），带来几个数量级的改进。优化分布式连结操作序列的一个基

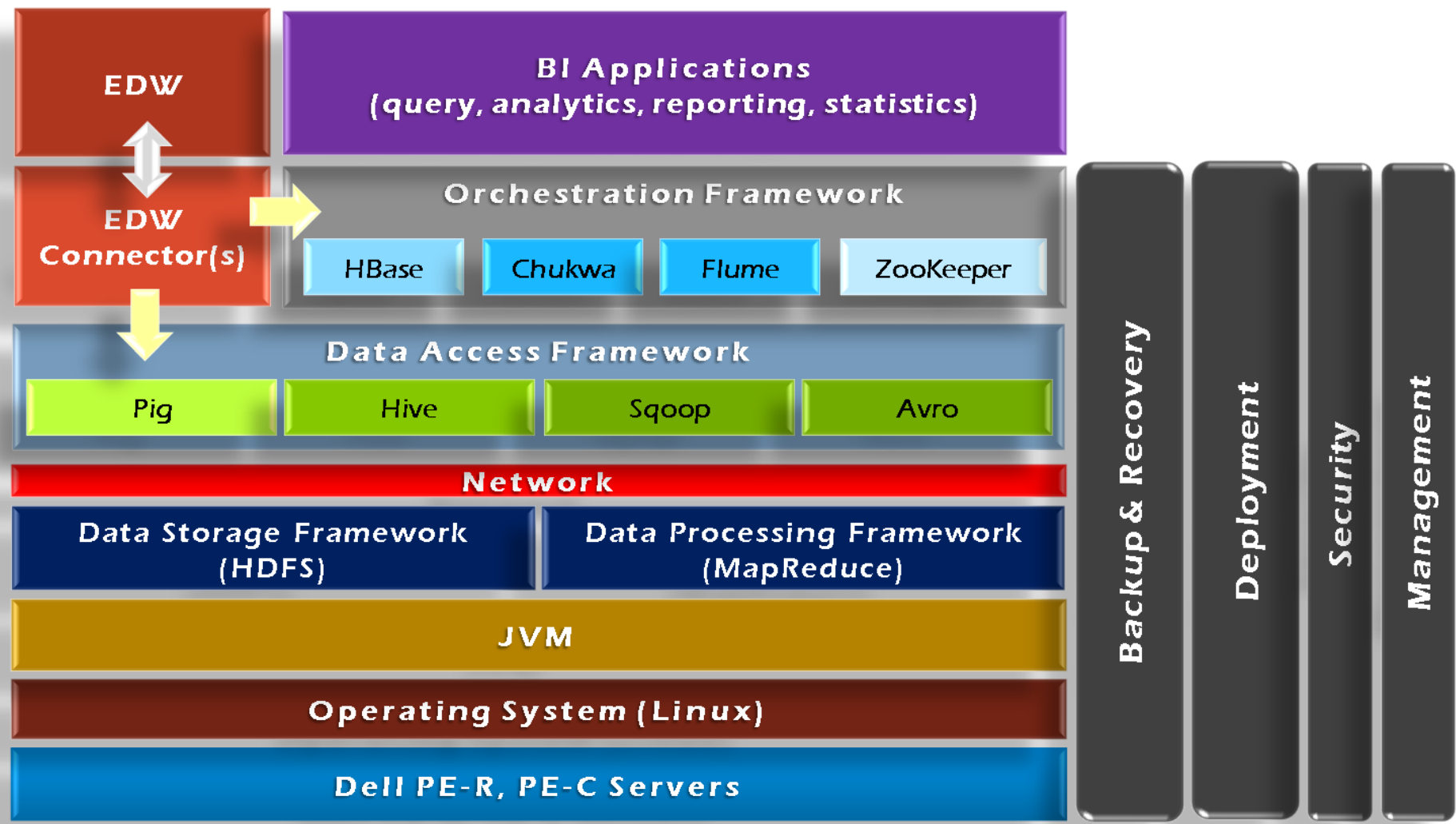
6.5.4 分布式查询执行

最后一层由拥有查询所需片段的所有站点执行局部查询（local query），要使用站点上的局部选择执行关系代数的算法。局部优化使用的都



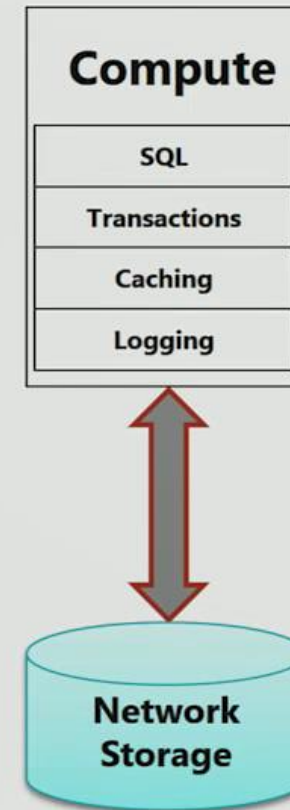
Now in **Big Data** & Cloud

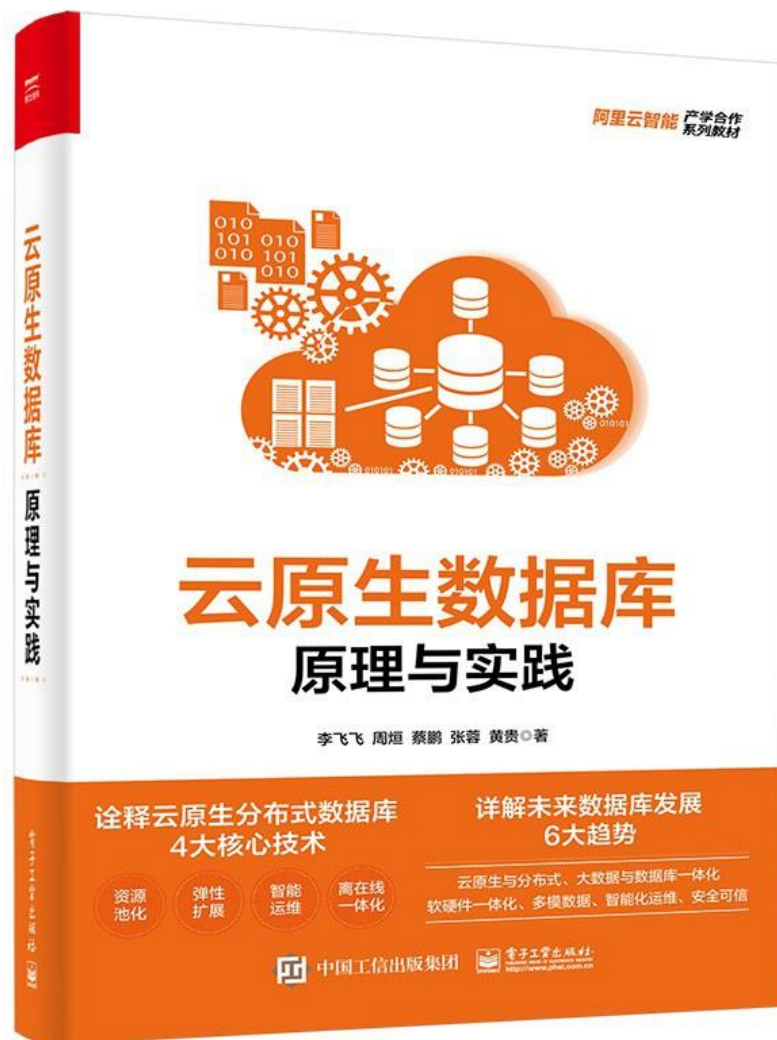




Database in cloud

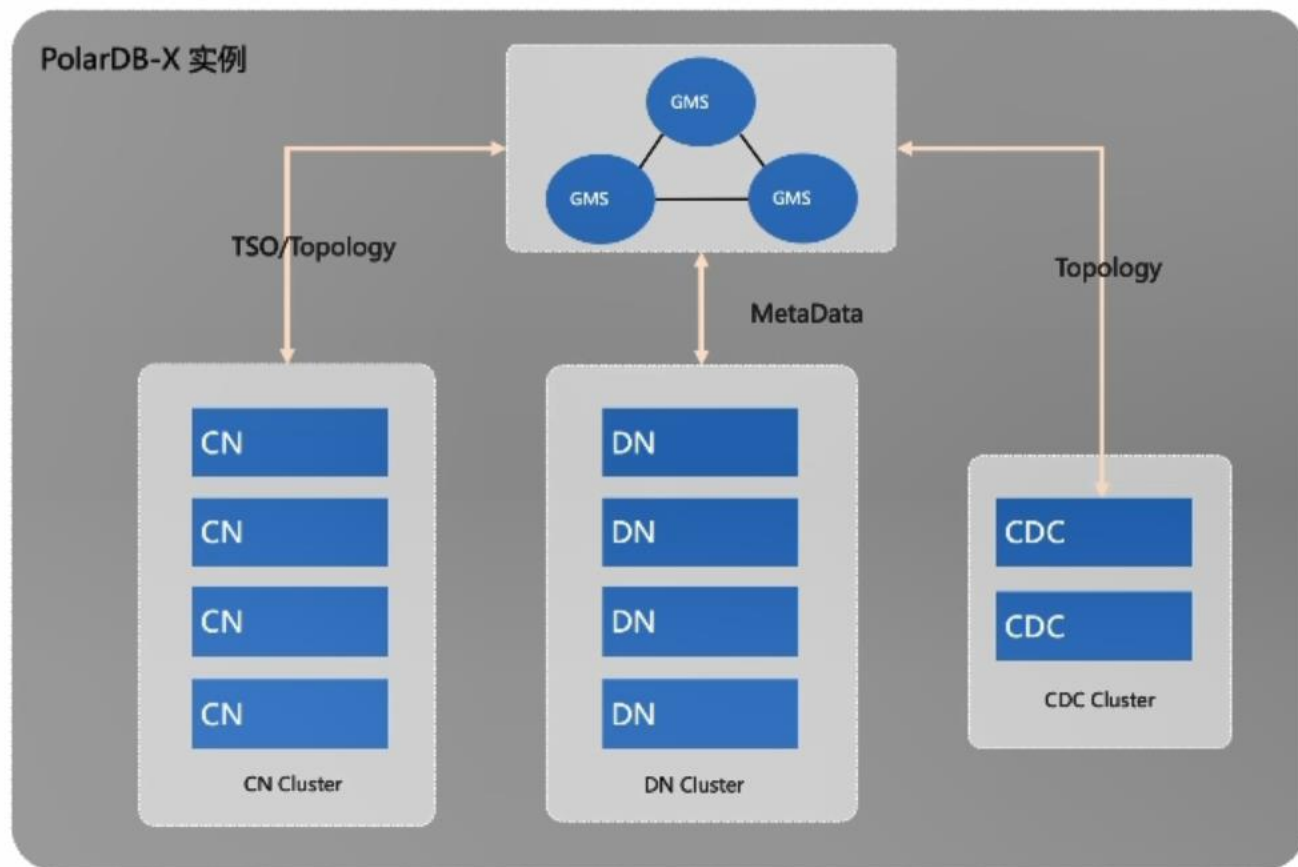
- Compute & Storage have different lifetimes
- Compute instances
 - fail and are replaced
 - are shut down to save cost
 - are scaled up/down/out based on load needs
- Storage on the other hand has to be long lived
- Decouple Compute and Storage for scalability, availability, durability





- 云原生数据库：原理与实践（全彩）（博文视点出品）
- 李飞飞，周烜，蔡鹏，张蓉，黄贵 ... 著
- 2022
- 电子工业出版社

PolarDB-X - 物理架构



元数据服务 (Global Meta Service, GMS)

- 提供全局授时服务 (TSO)
- 维护 Table/Schema、Statistic 等 Meta 信息
- 维护账号、权限等安全信息

存储节点 (Data Node, DN)

- 基于多数派 Paxos 共识协议的高可靠存储
- 处理分布式 MVCC 事务的可见性判断

计算节点 (Compute Node, CN)

- 基于无状态的 SQL 引擎提供分布式路由和计算
- 处理分布式事务的 2PC 协调、全局索引维护等

日志节点 (Change Data Capture, CDC)

- 提供兼容 MySQL 生态的 binlog 协议和数据格式
- 提供兼容 MySQL Replication 主从复制的交互

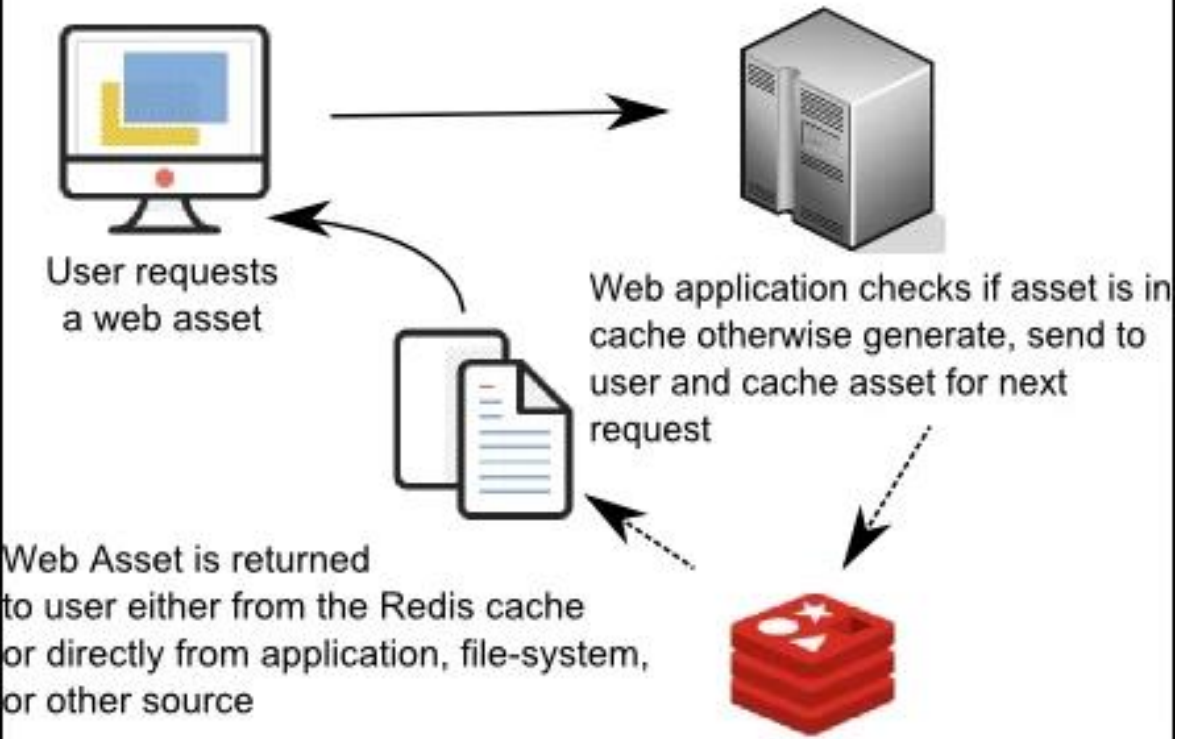
Even distributed cache

□ Redis

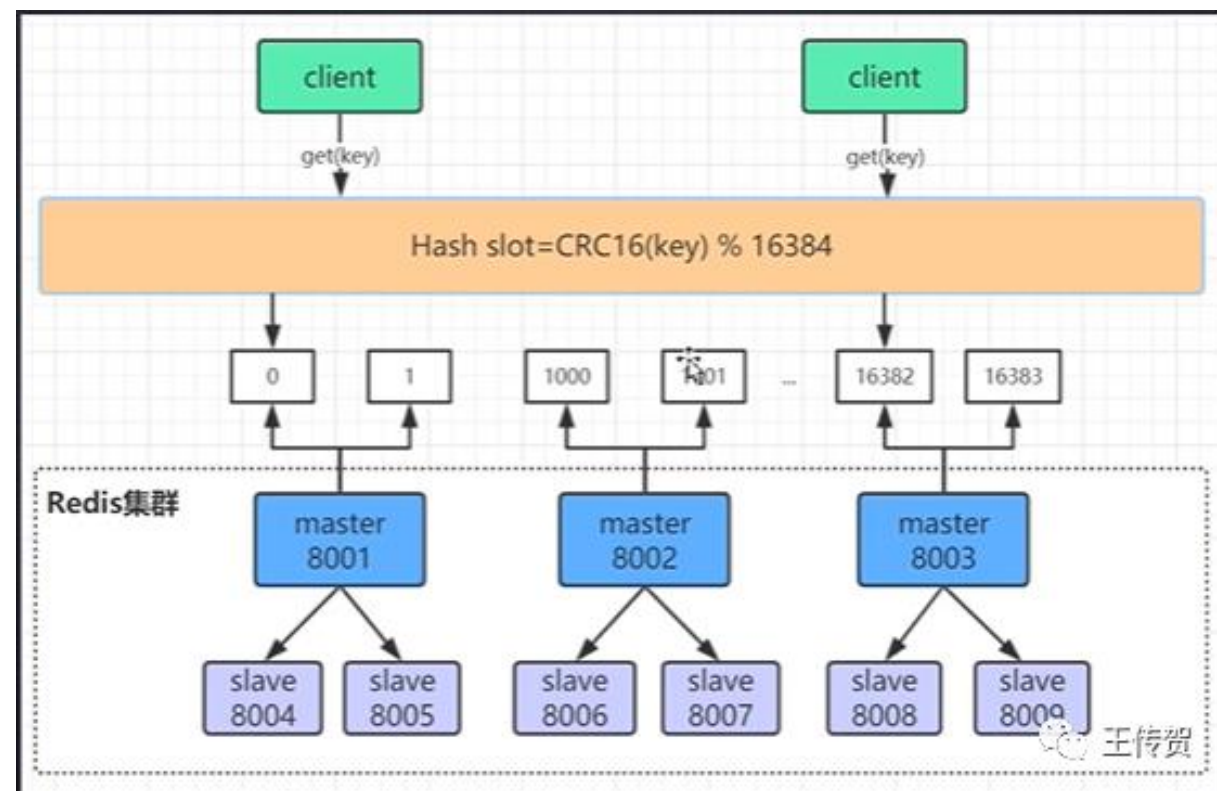
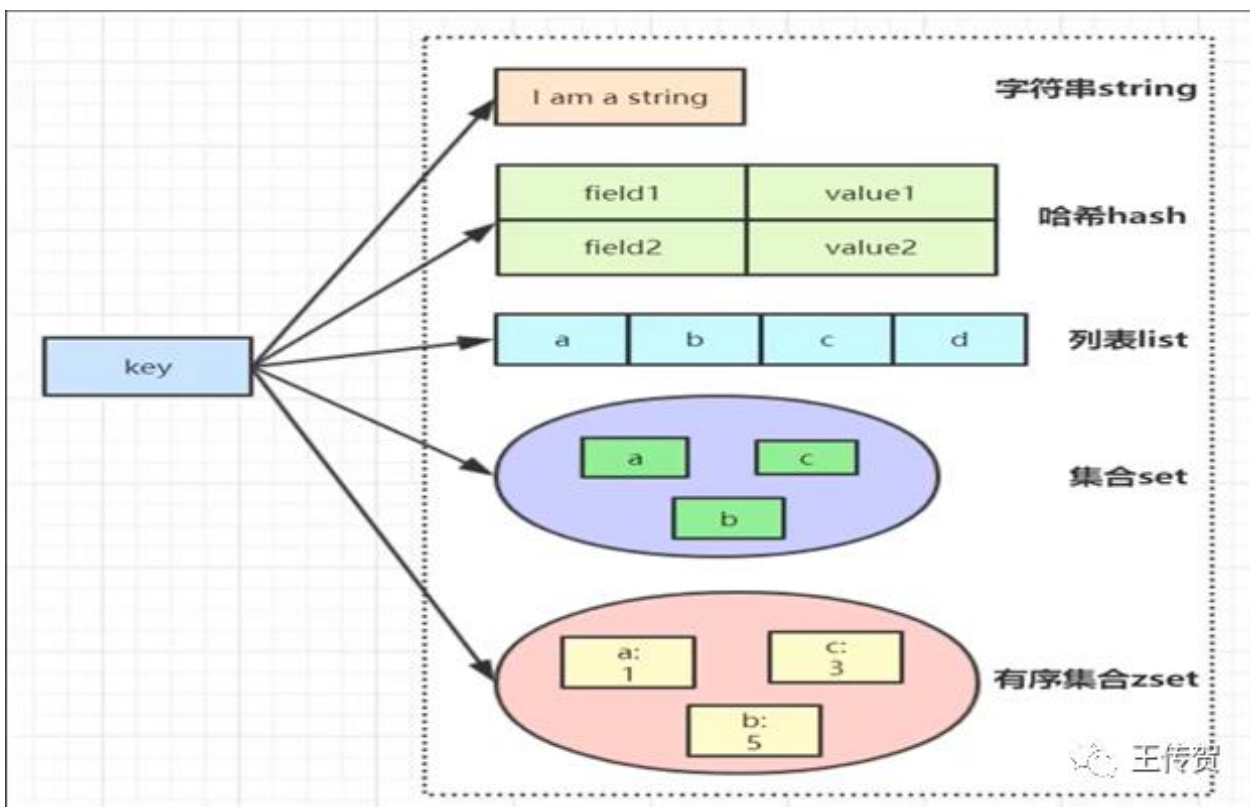
- 某企业是为城市高端用户提供高品质蔬菜生鲜服务的初创企业，创业初期为快速开展业务，该企业采用轻量型的开发架构（脚本语言+关系型数据库）研制了一套业务系统。业务开展后受到用户普遍欢迎，用户数和业务数量迅速增长，原有的数据库服务器已不能满足高度并发的业务要求。为此，该企业成立了专门的研发团队来解决该问题。
- 张工建议重新开发整个系统，采用新的服务器和数据架构，解决当前问题的同时为日后的扩展提供支持。但是，李工认为张工的方案开发周期过长，投入过大，当前应该在改动尽量小的前提下解决该问题。李工认为访问量很大的只是部分数据，建议采用缓存工具MemCache来减轻数据库服务器的压力，这样开发量小，开发周期短，比较适合初创公司，同时将来也可以通过集群进行扩展。然而，刘工又认为李工的方案中存在数据可靠性和一致性问题，在宕机时容易丢失交易数据，建议采用Redis来解决问题。在经过充分讨论，该公司最终决定采用刘工的方案



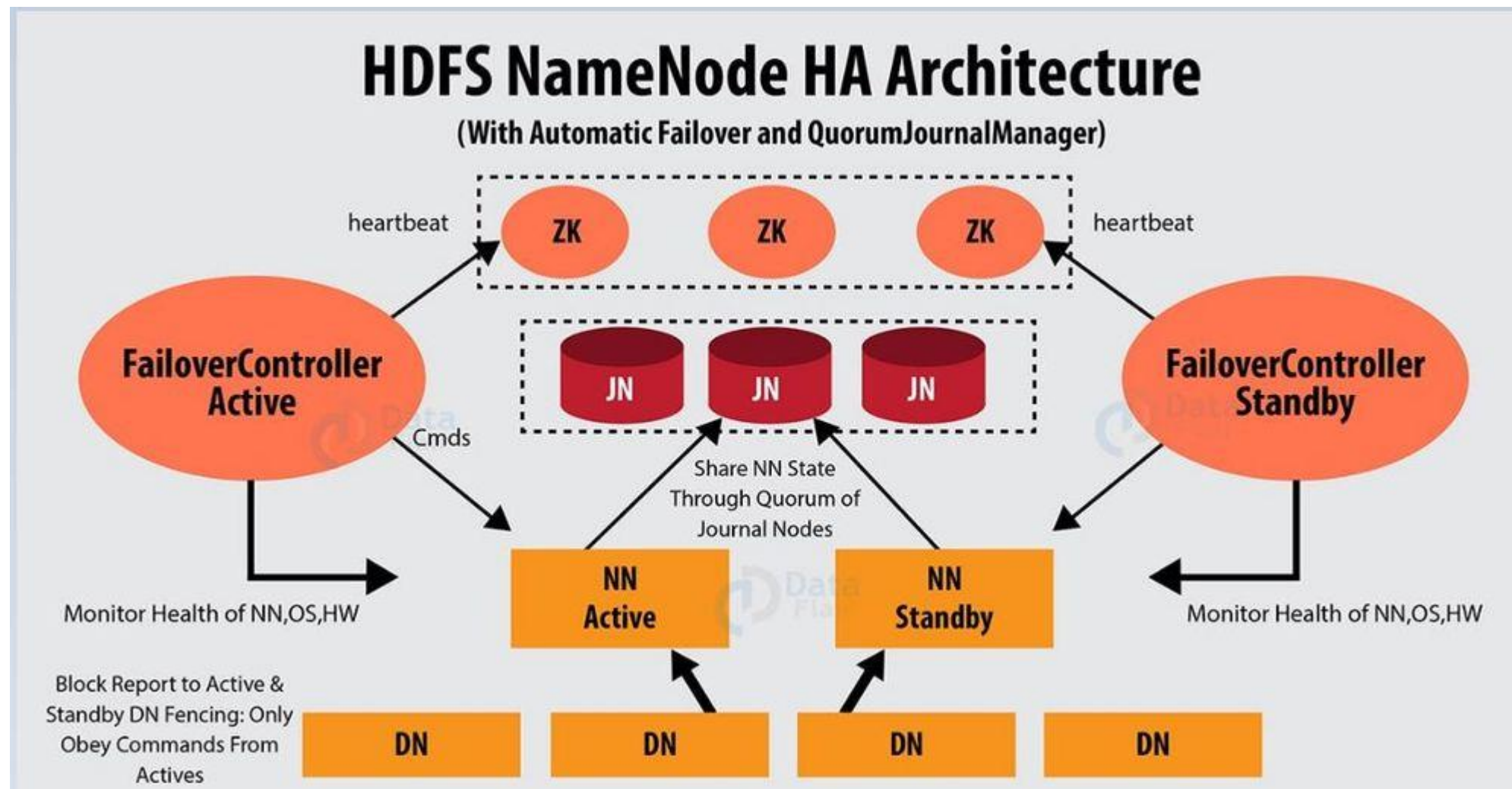
Redis as a Web Cache



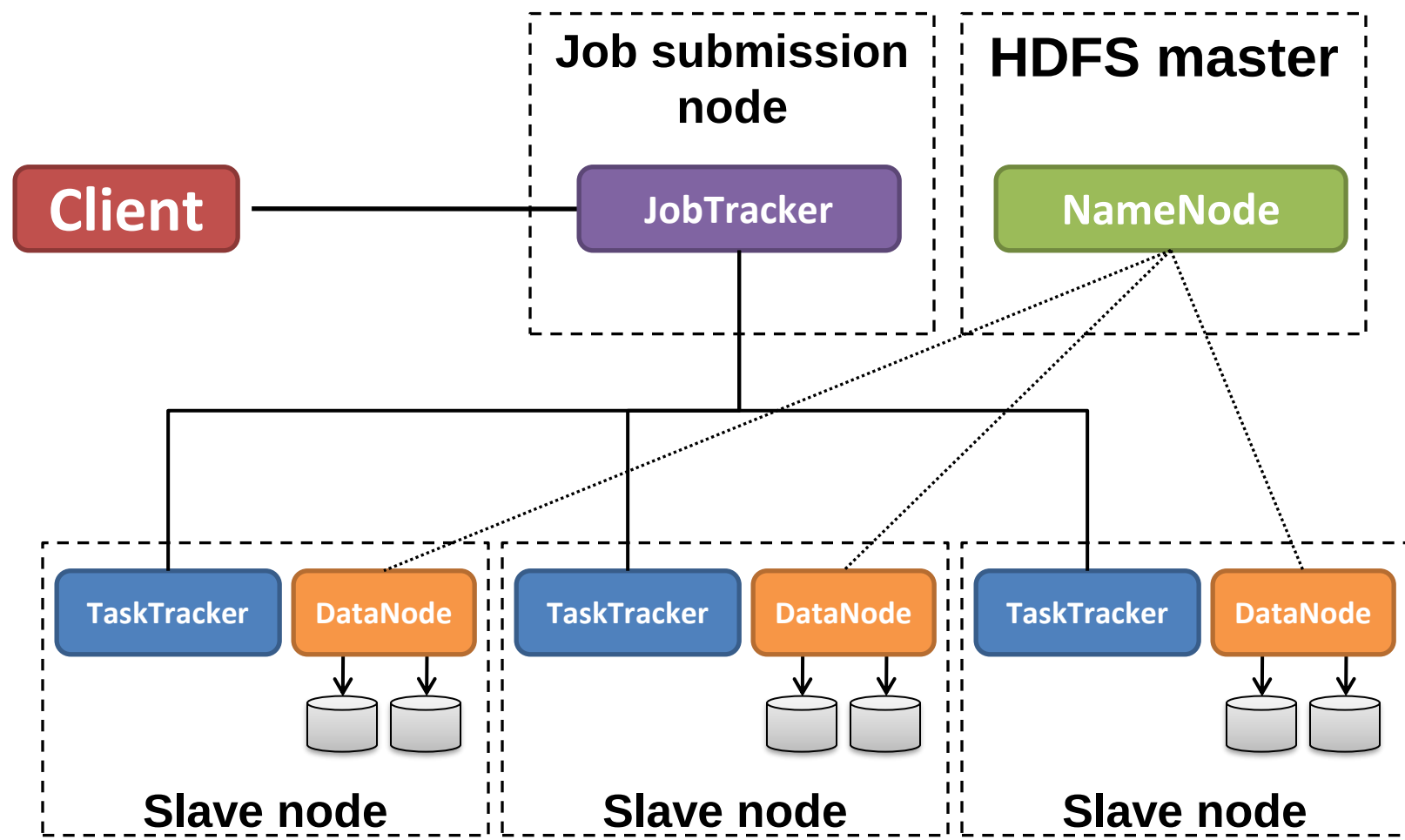
- ❑ **A very popular use pattern for Redis is as an in-memory cache for web applications.**
 - Redis is available as a caching option for popular web frameworks such as Django, Ruby-on-Rails, Node.js, and Flask.
 - As a popular caching technology Redis excels in web applications for storing new data while evicting stale data.
 - For web applications, the cached data can range from single HTML character strings, widgets, and elements to entire web pages and websites.

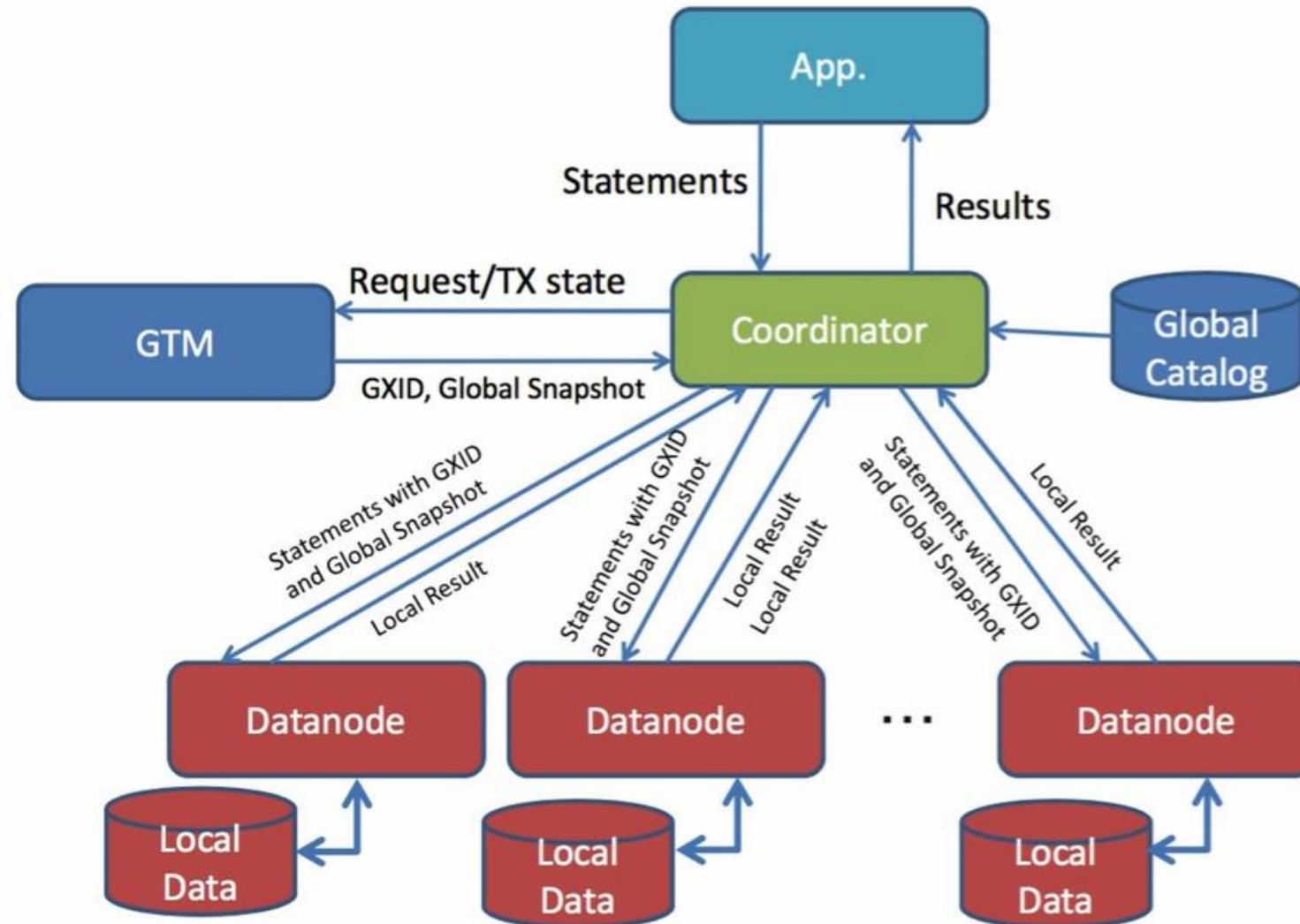


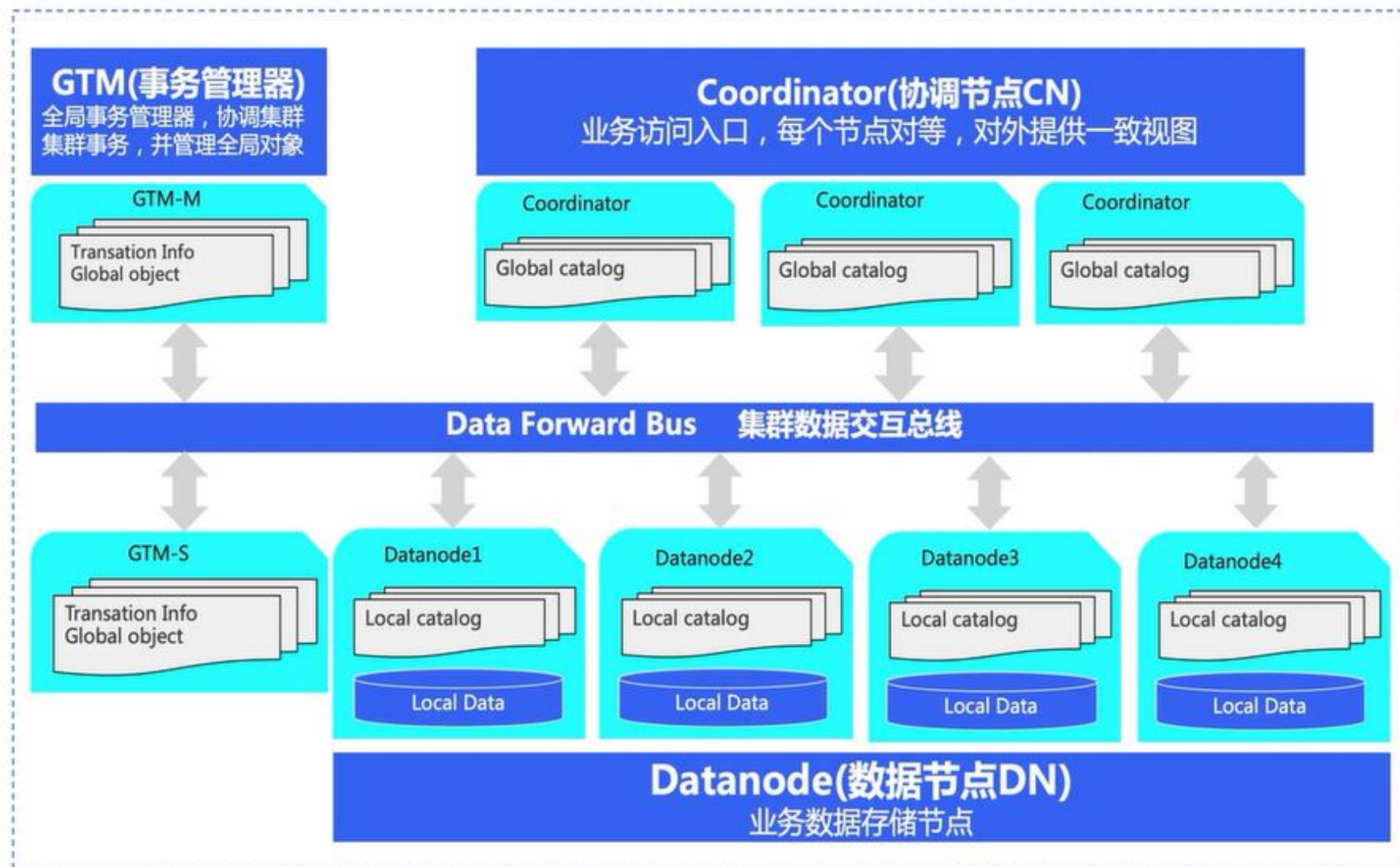
They share a lot!



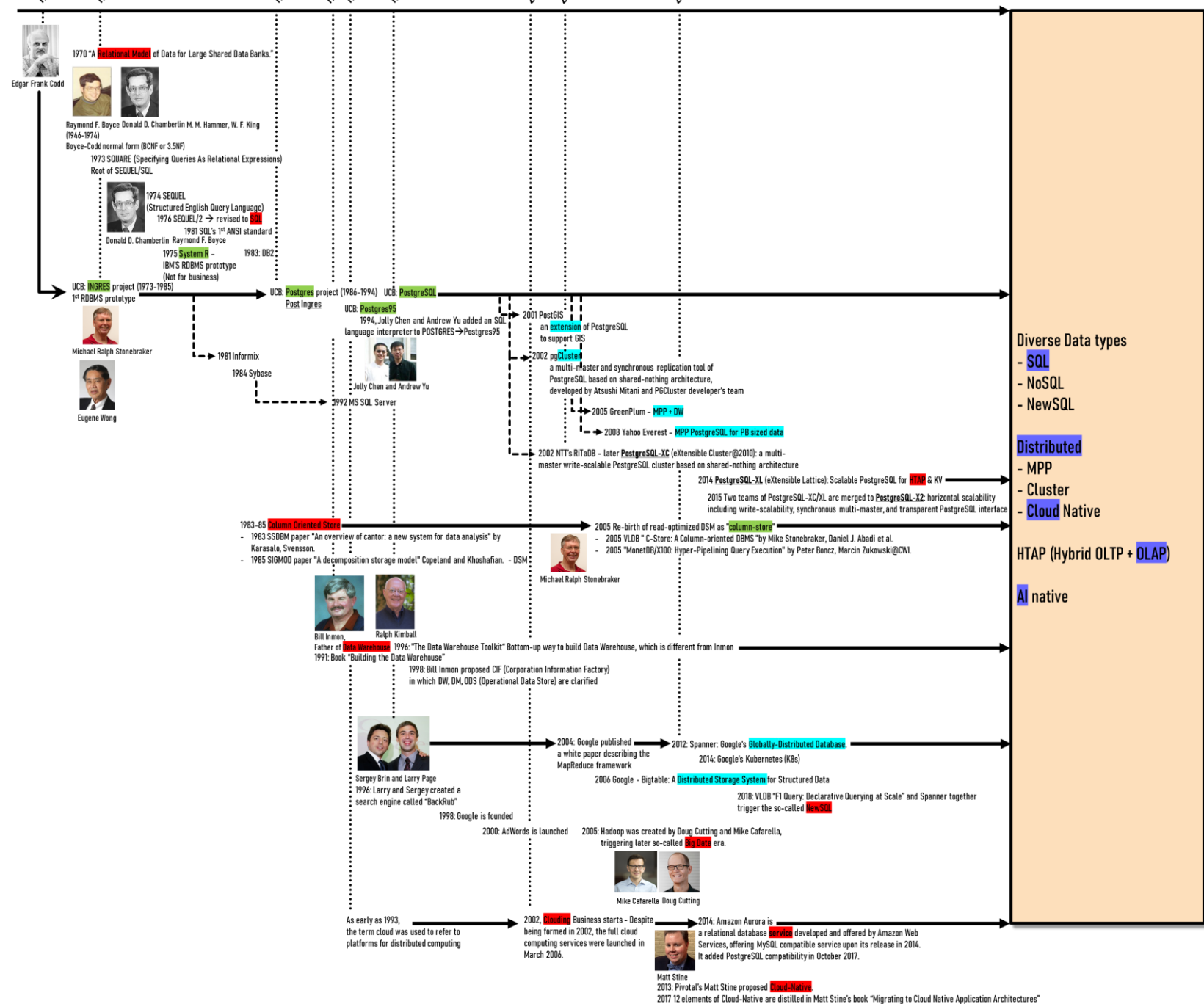
大数据软件也支持一些编程 – 如 Hadoop 上的 MR



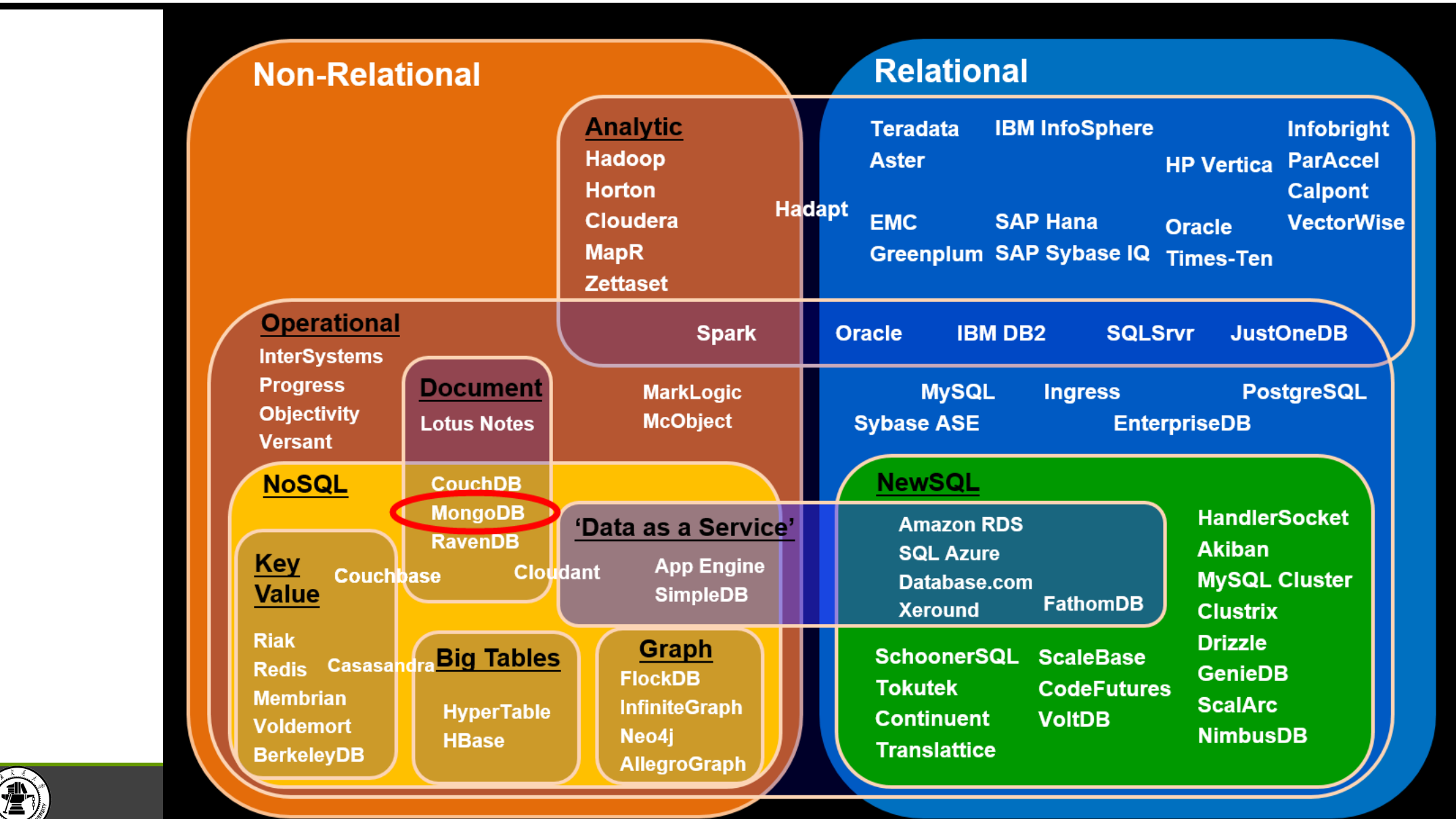




PostgreSQL的外溢，也是受其他领域的影响 – 分布式 (MPP, Cluster), 大数据, 云, AI等



Non-Relational vs. Relational



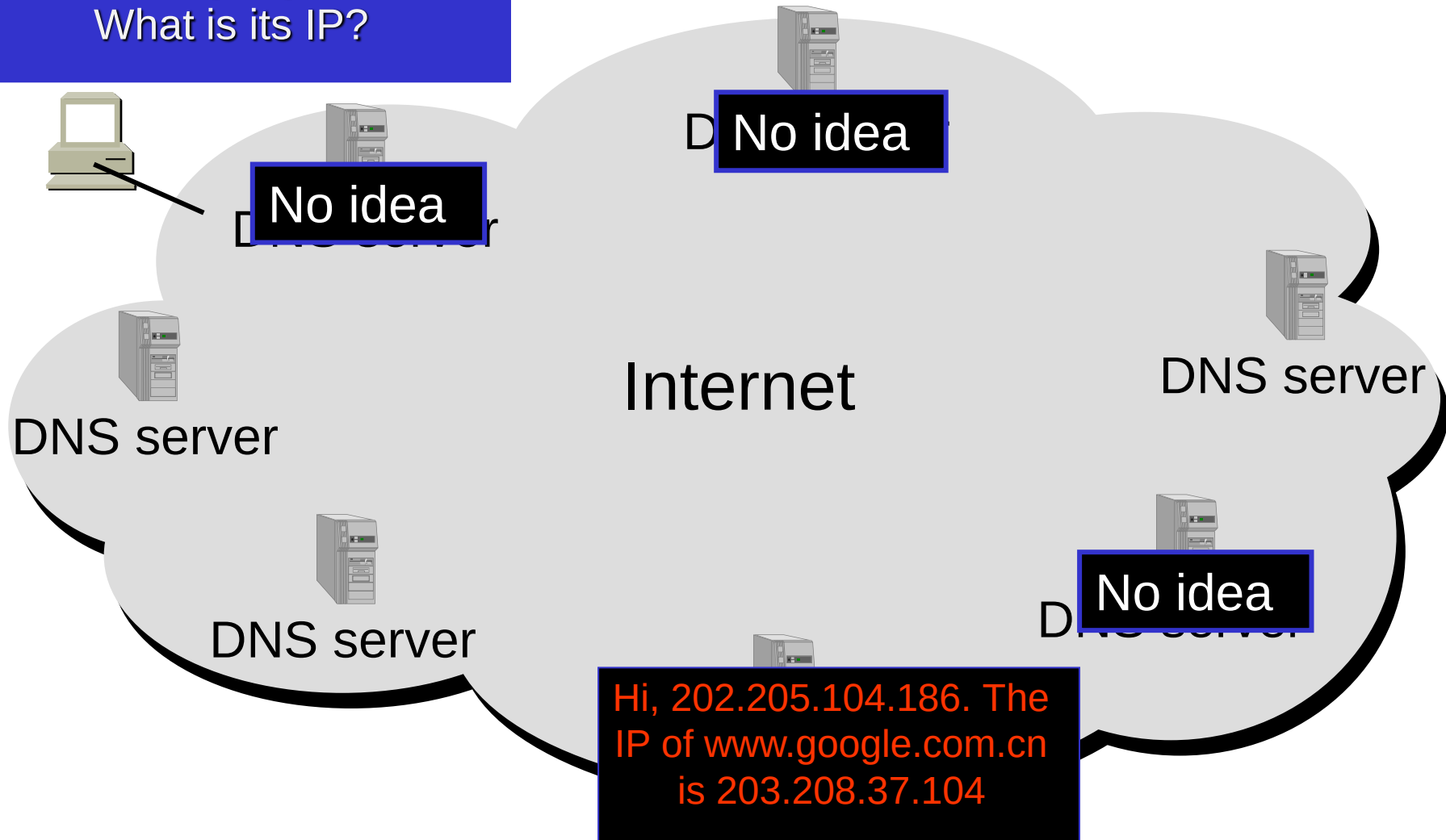
P2P? – DNS system – 1983

□ DNS - Domain Name Service

- Proposed in 1983 by Paul Mockapetris
- Aims to assign IP with semantic meaning

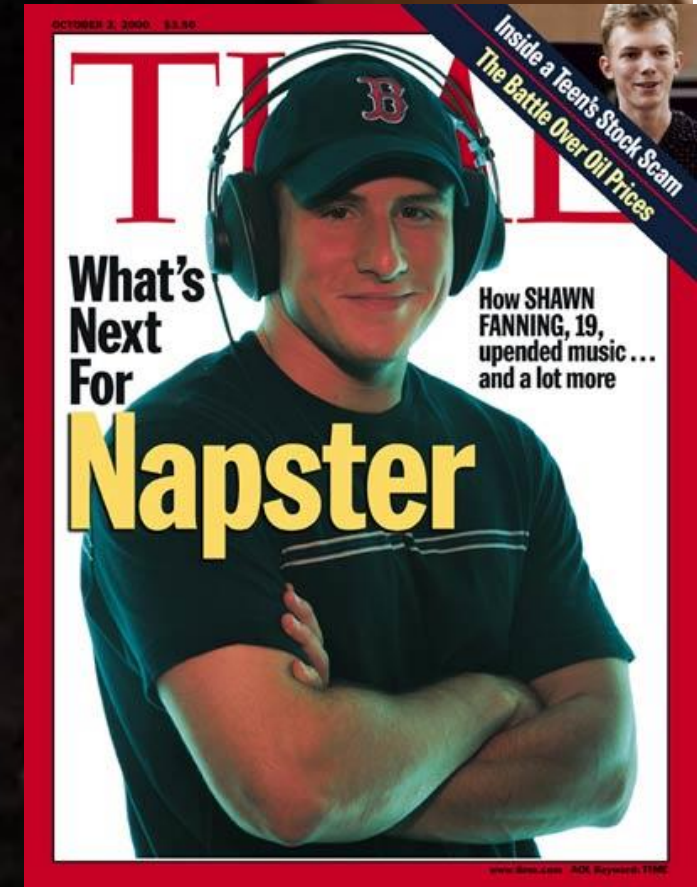


I am 202.205.104.186. I want
to visit www.google.com.cn.
What is its IP?



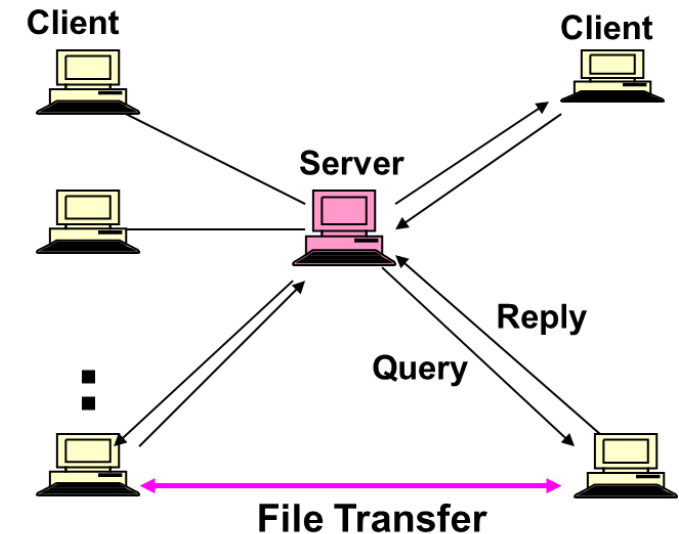
P2P is popular

- 1999 Napster to share music

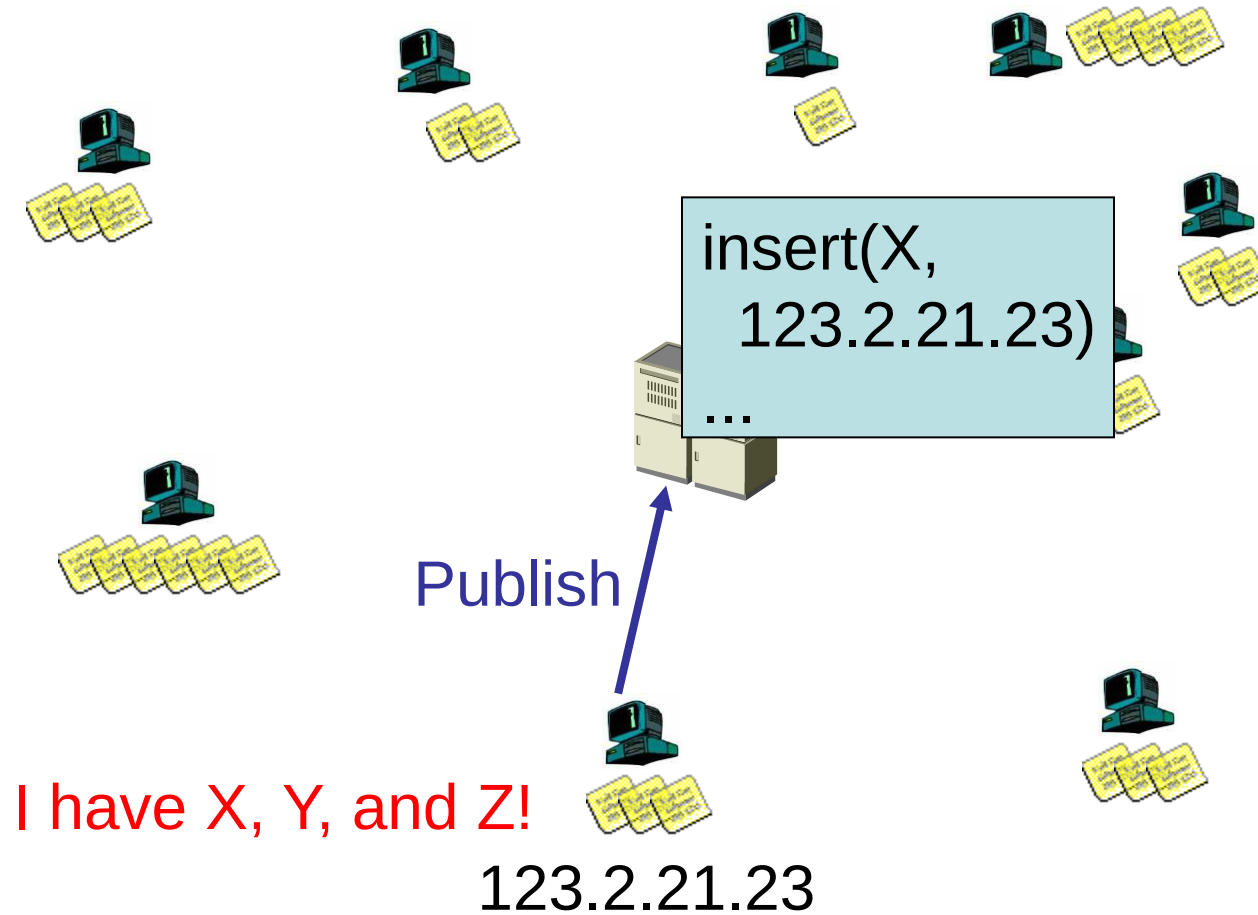


Shawn Fanning

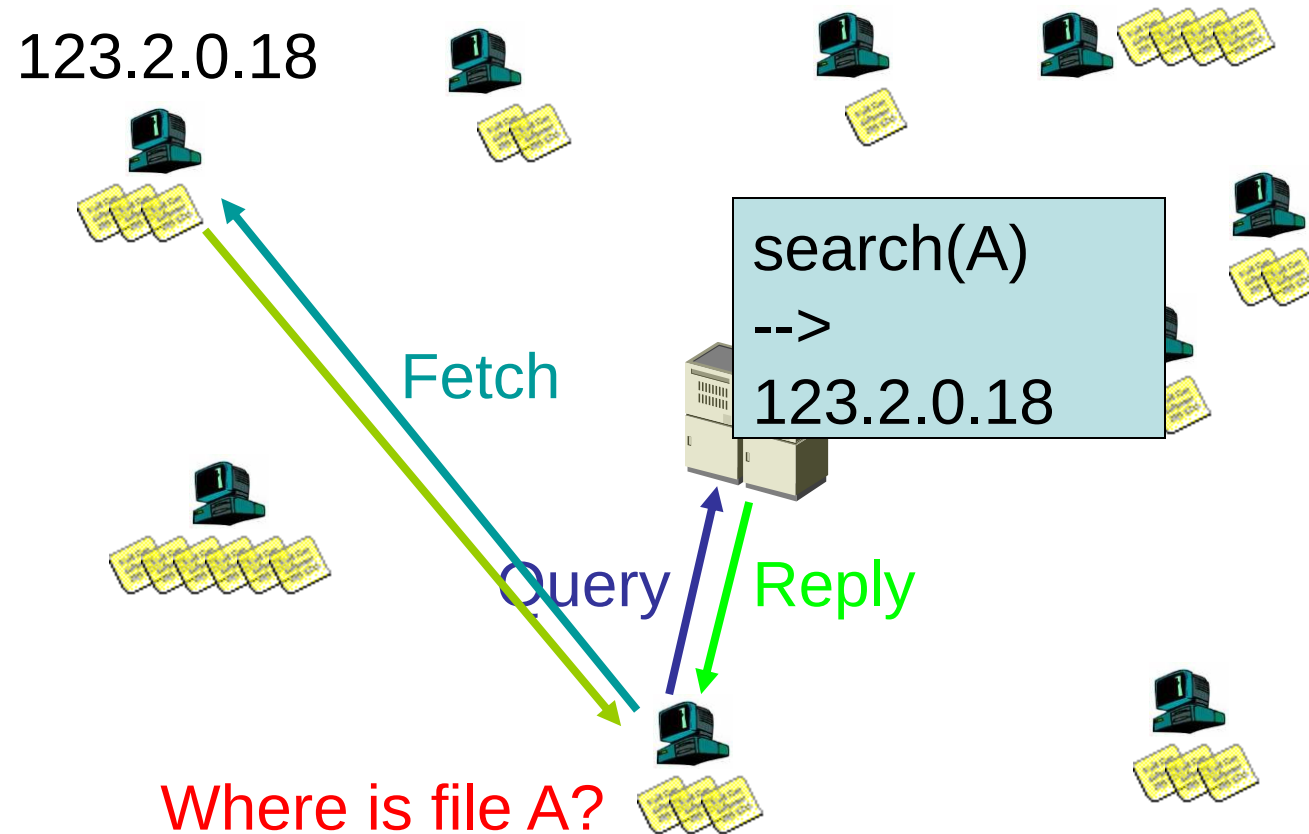
- Share Music files, MP3 data
- Nodes register their contents (list of files) and IPs with server
- Centralized server for searches
 - The client sends queries to the centralized server for files of interest
 - Keyword search (artist, song, album, bitrate, etc.)
- Napster server replies with IP address of users with matching files
- File download done on a peer to peer basis
- Poor scalability
- Single point of failure
- Legal issues → shutdown



Napster: Publish



Napster: Search



Chord: Distributed Lookup (Directory) Service

□ Key design decision

- Decouple correctness from efficiency

□ Properties

- Each node needs to know about $O(\log(M))$, where M is the total number of nodes
- Guarantees that a tuple is found in $O(\log(M))$ steps

□ Many other lookup services: CAN, Tapestry, Pastry, Kademlia, ...

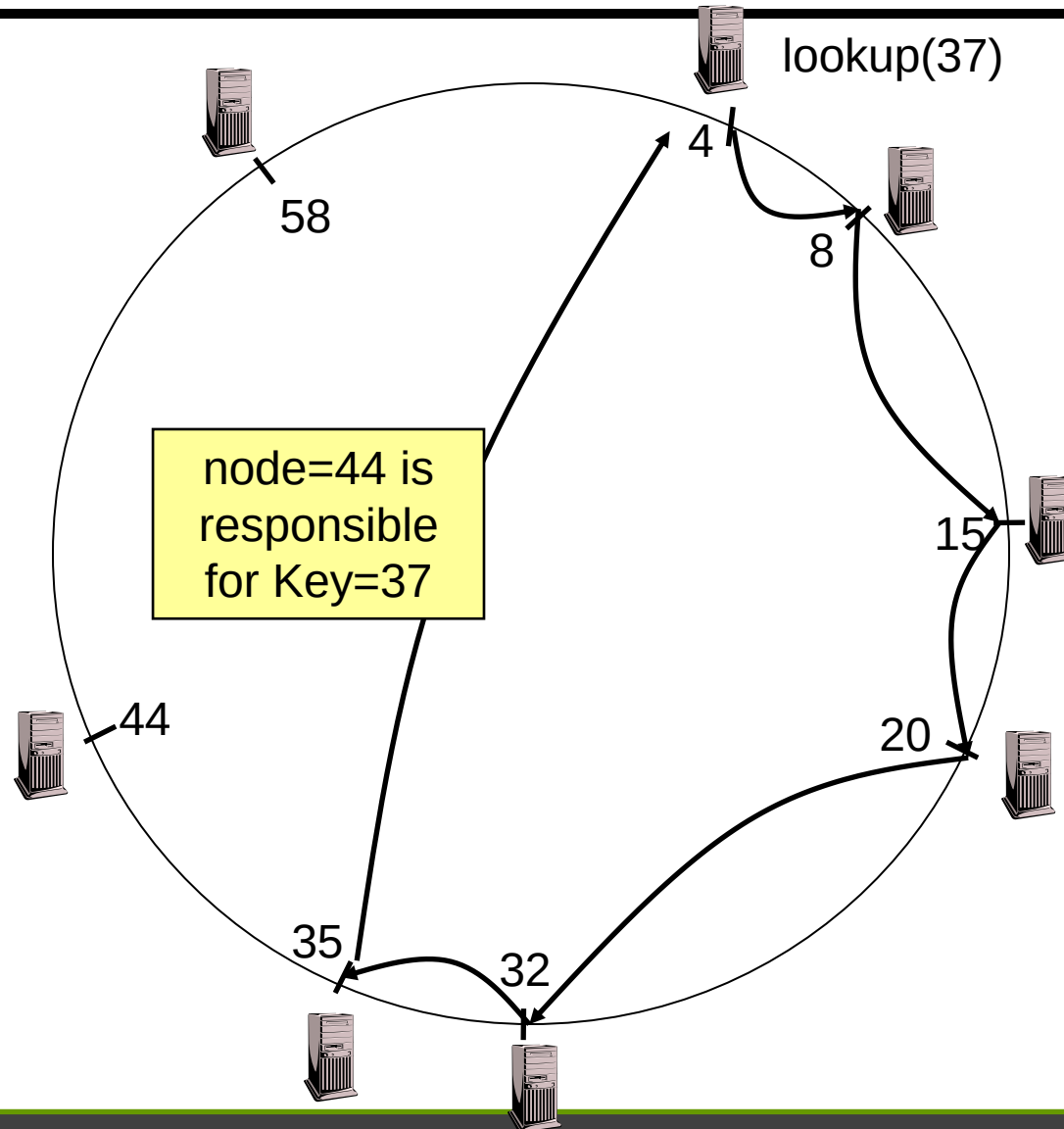


Lookup

- Each node maintains pointer to its successor

- Route packet (Key, Value) to the node responsible for ID using successor pointers

- E.g., node=4 lookups for node responsible for Key=37



Stabilization Procedure

- Periodic operation performed by each node n to maintain its successor when new nodes join the system

```
n.stabilize()
  x = succ.pred;
  if (x  $\lvert$  (n, succ))
    succ = x;    // if x better successor, update
  succ.notify(n); // n tells successor about itself

n.notify(n')
  if (pred = nil or n'  $\lvert$  (pred, n))
    pred = n';    // if n' is better predecessor, update
```


Joining Operation

Node with id=50 joins the ring

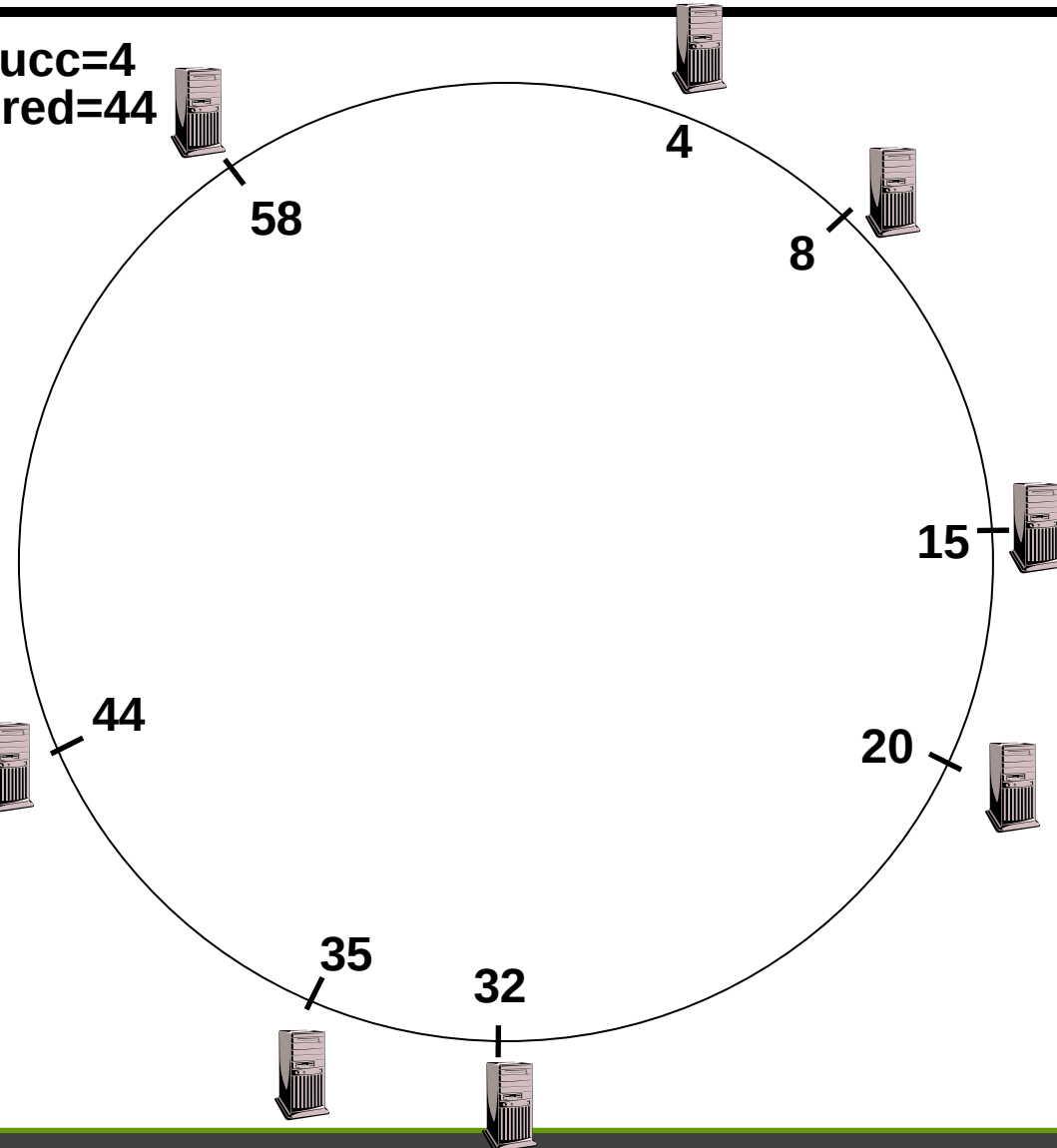
Node 50 needs to know at least one node already in the system

- Assume known node is 15

succ=nil
pred=nil
50

succ=58
pred=35
44

succ=4
pred=44
58

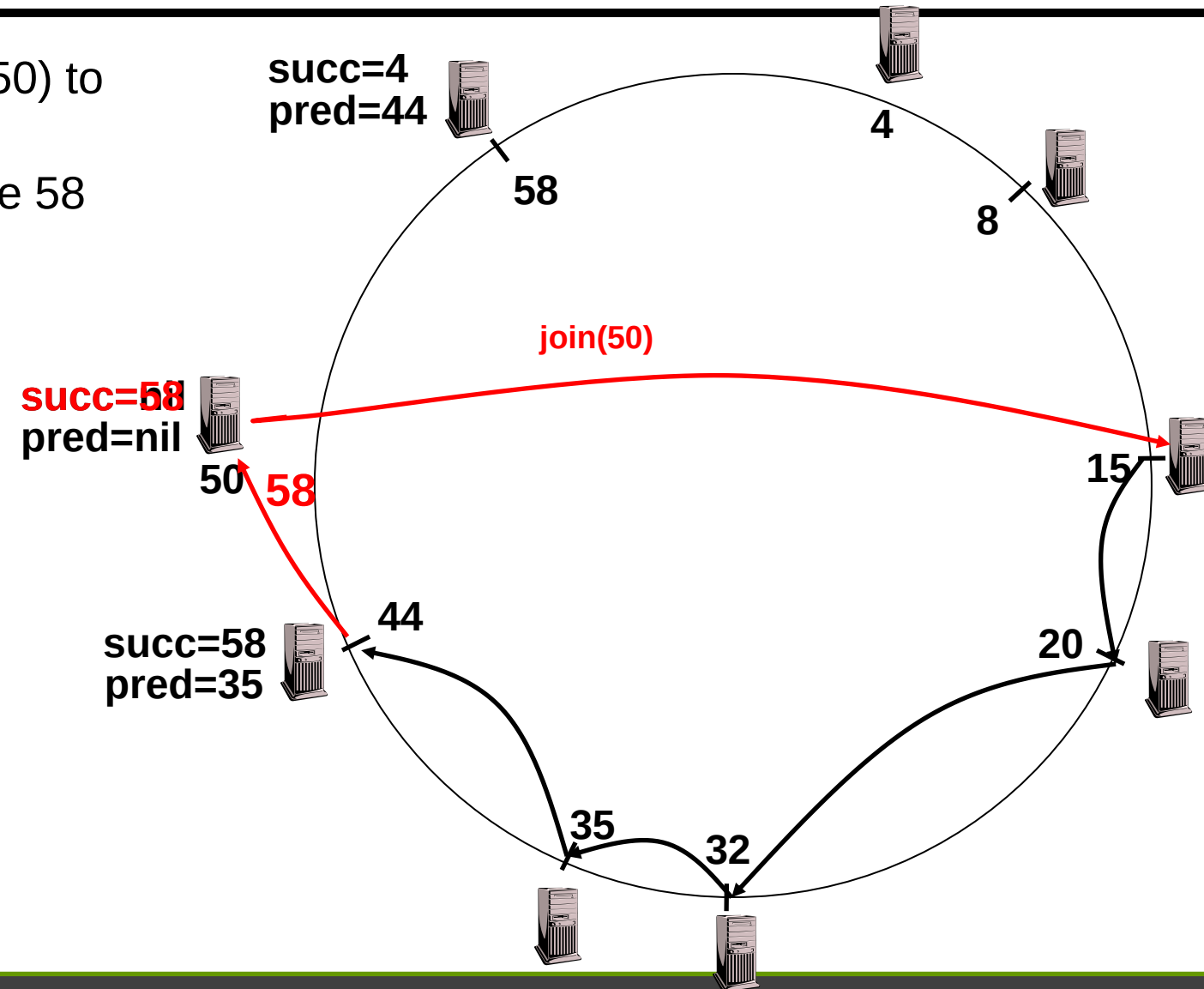


Joining Operation

n=50 sends join(50) to
node 15

n=44 returns node 58

n=50 updates its
successor to 58



Joining Operation

n=50 executes
stabilize()

n's successor (58)
returns x = 44

succ=58
pred=nil

x=44

succ=4
pred=44

succ=58
pred=35

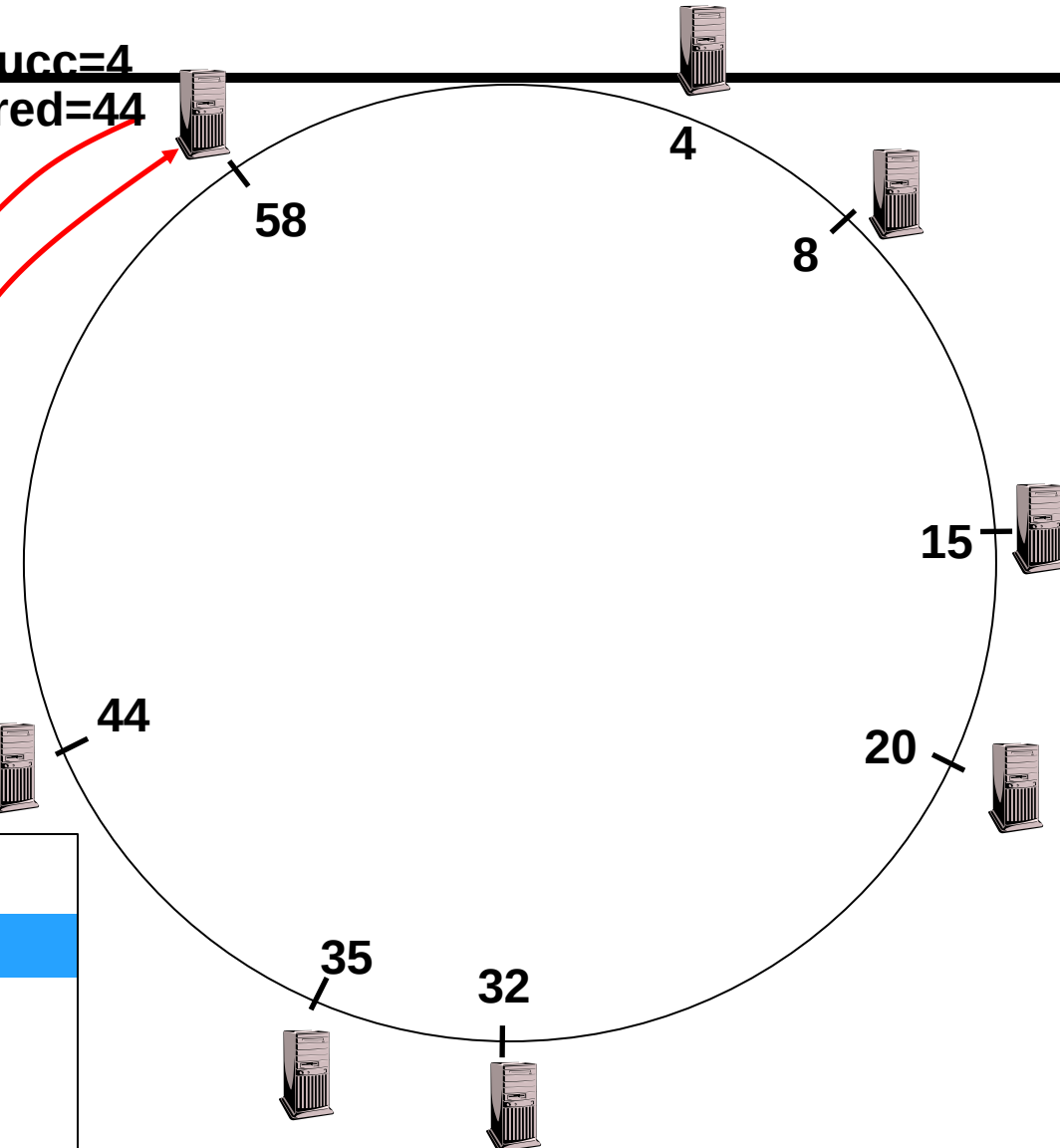
n.stabilize()

x = succ.pred;

if (x | (n, succ))

 succ = x;

 succ.notify(n);



Joining Operation

n=50 executes
stabilize()

x = 44

succ = 58

succ=58
pred=nil

50

succ=58
pred=35

44

n.stabilize()

x = succ.pred;

if (x | (n, succ))

succ = x;

succ.notify(n);

succ=4
pred=44

58

4

8

15

20

35

32

Joining Operation

n=50 executes
stabilize()

x = 44

succ = 58

n=50 sends to it's
successor (58)
notify(50)

succ=58
pred=nil

50

succ=58
pred=35

44

n.stabilize()

x = succ.pred;

if (x | (n, succ))

succ = x;

succ.notify(n);

succ=4
pred=44

58

4

8

15

20

35

32

notify(50)



Joining Operation

n=58 processes

notify(50)

pred = 44

n' = 50

succ=58
pred=nil

50

succ=58
pred=35

44

succ=4
pred=44

58

4

8

15

20

35

32

n.notify(n')

if (pred = nil or n' l(pred, n))

pred = n'

notify(50)

Joining Operation

n=58 processes

notify(50)

pred = 44

n' = 50

set pred = 50

succ=58
pred=nil

50

succ=58
pred=35

44

succ=4
pred=50

58

4

8

15

20

35

32

n.notify(n')

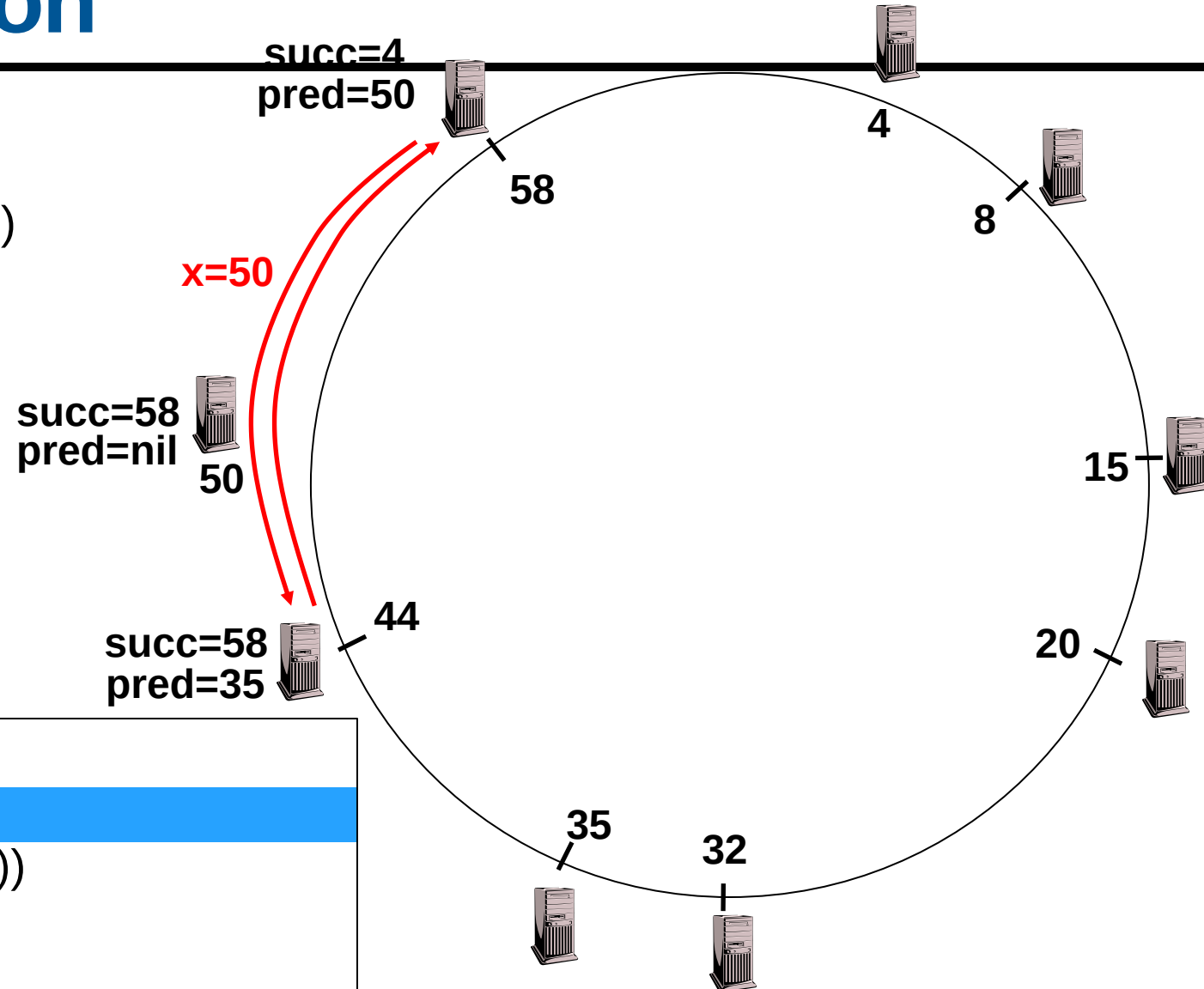
if (pred = nil or n' l(pred, n))

pred = n'

Joining Operation

n=44 runs
stabilize()

n's successor (58)
returns x = 50



n.stabilize()

x = succ.pred;

if (x | (n, succ))

succ = x;

succ.notify(n);

Joining Operation

n=44 runs
stabilize()

x = 50

succ = 58

succ=58
pred=nil

50

succ=58
pred=35

44

n.stabilize()

x = succ.pred;

if (x | (n, succ))

succ = x;

succ.notify(n);

succ=4
pred=50

58

4

8

15

20

35

32

Joining Operation

n=44 runs
stabilize()

x = 50

succ = 58

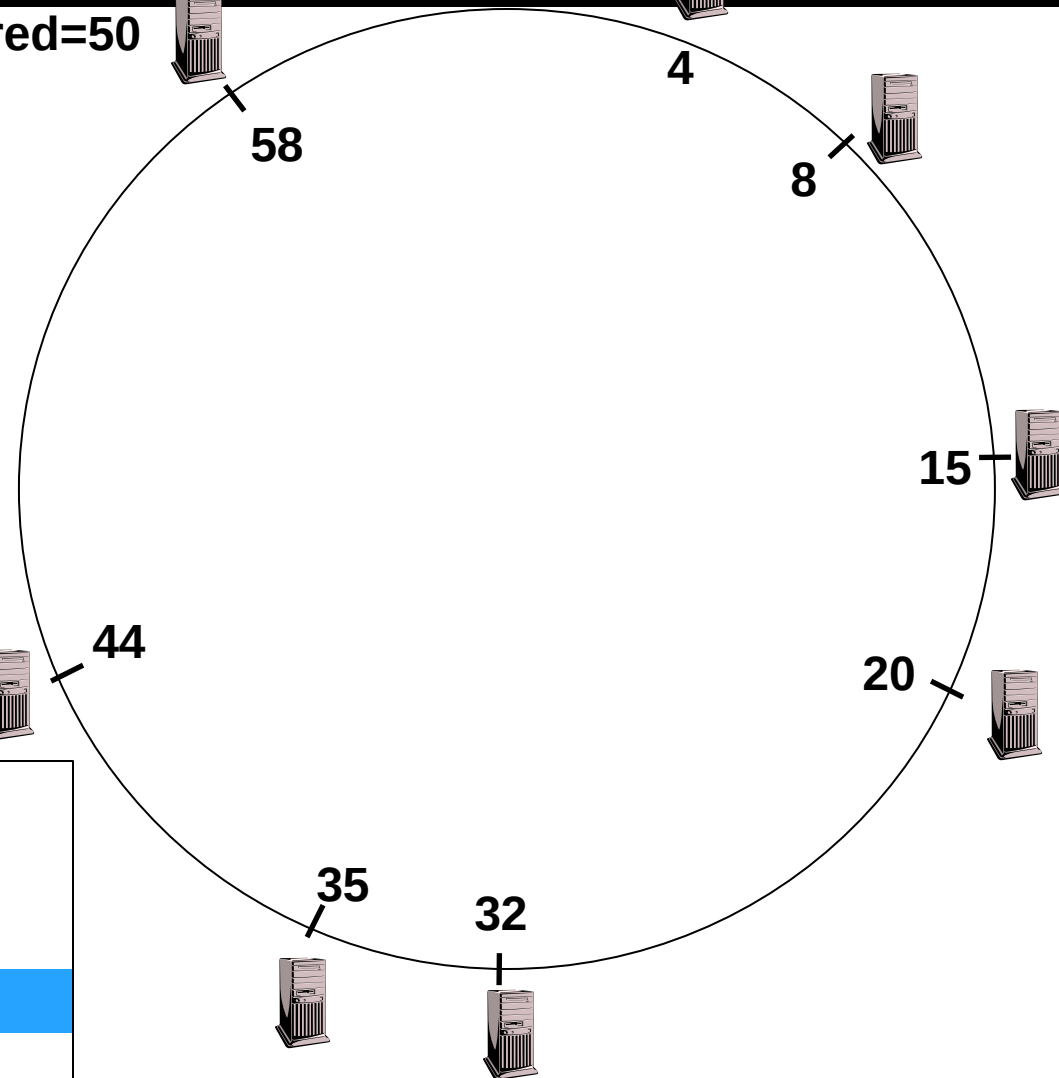
n=44 sets succ=50

succ=58
pred=nil
50

~~succ=58~~
pred=35
44

n.stabilize()
x = succ.pred;
if (x | (n, succ))
succ = x;
succ.notify(n);

succ=4
pred=50



Joining Operation

n=44 runs stabilize()

n=44 sends
notify(44) to its
successor

succ=58
pred=nil

50

notify(44)

succ=50
pred=35

44

n.stabilize()

x = succ.pred;

if (x | (n, succ))

succ = x;

succ.notify(n);

succ=4
pred=50

58

4

8

15

20

35

32

Joining Operation

n=50 processes
notify(44)
pred = nil

succ=4
pred=50

succ=58
pred=nil

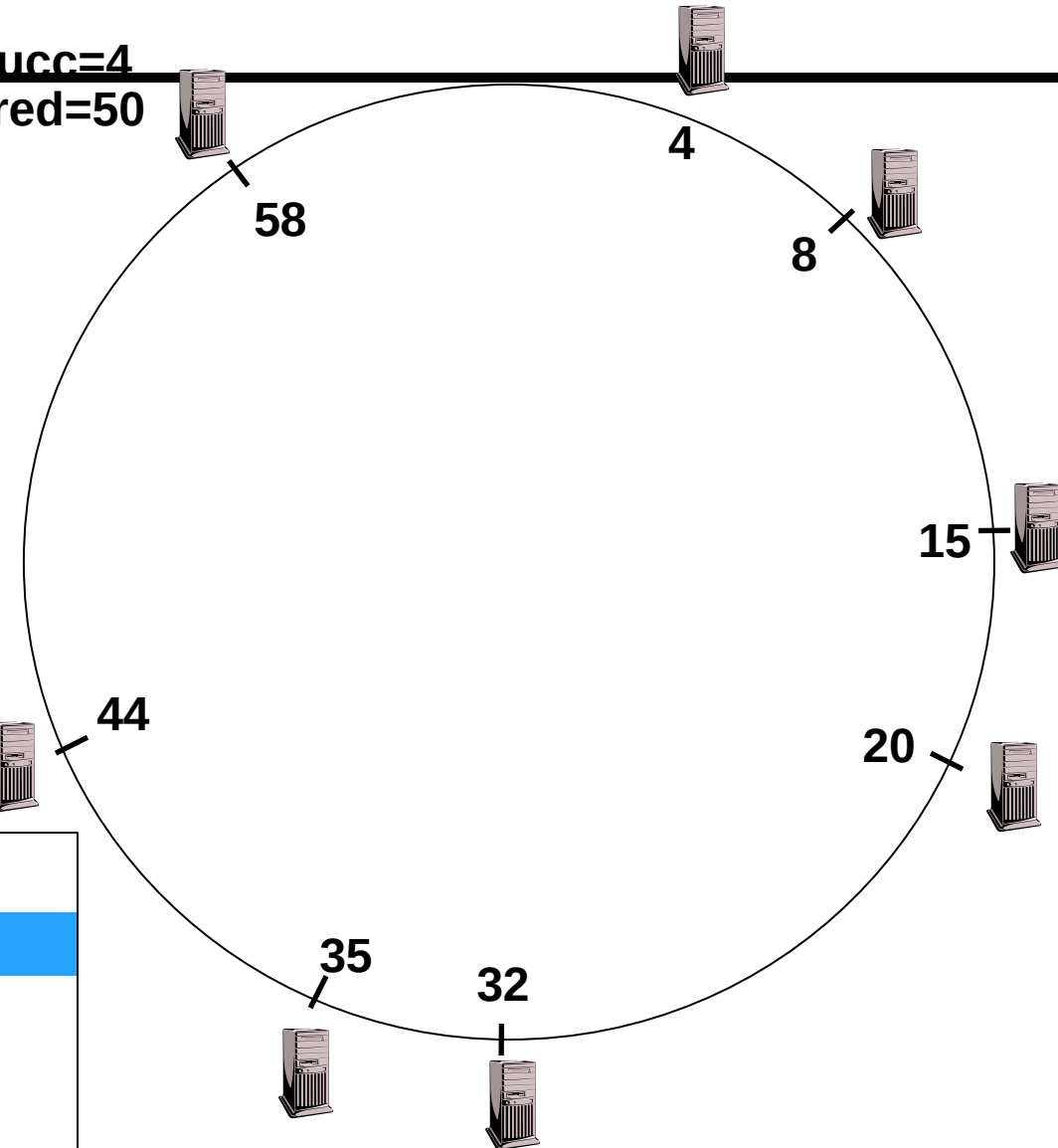
notify(44)

succ=50
pred=35

n.notify(n')

if (pred = nil or n' l(pred, n))

pred = n'



Joining Operation

n=50 processes

notify(44)

pred = nil

n=50 sets pred=44

succ=4
pred=50

succ=58
~~pred=44~~

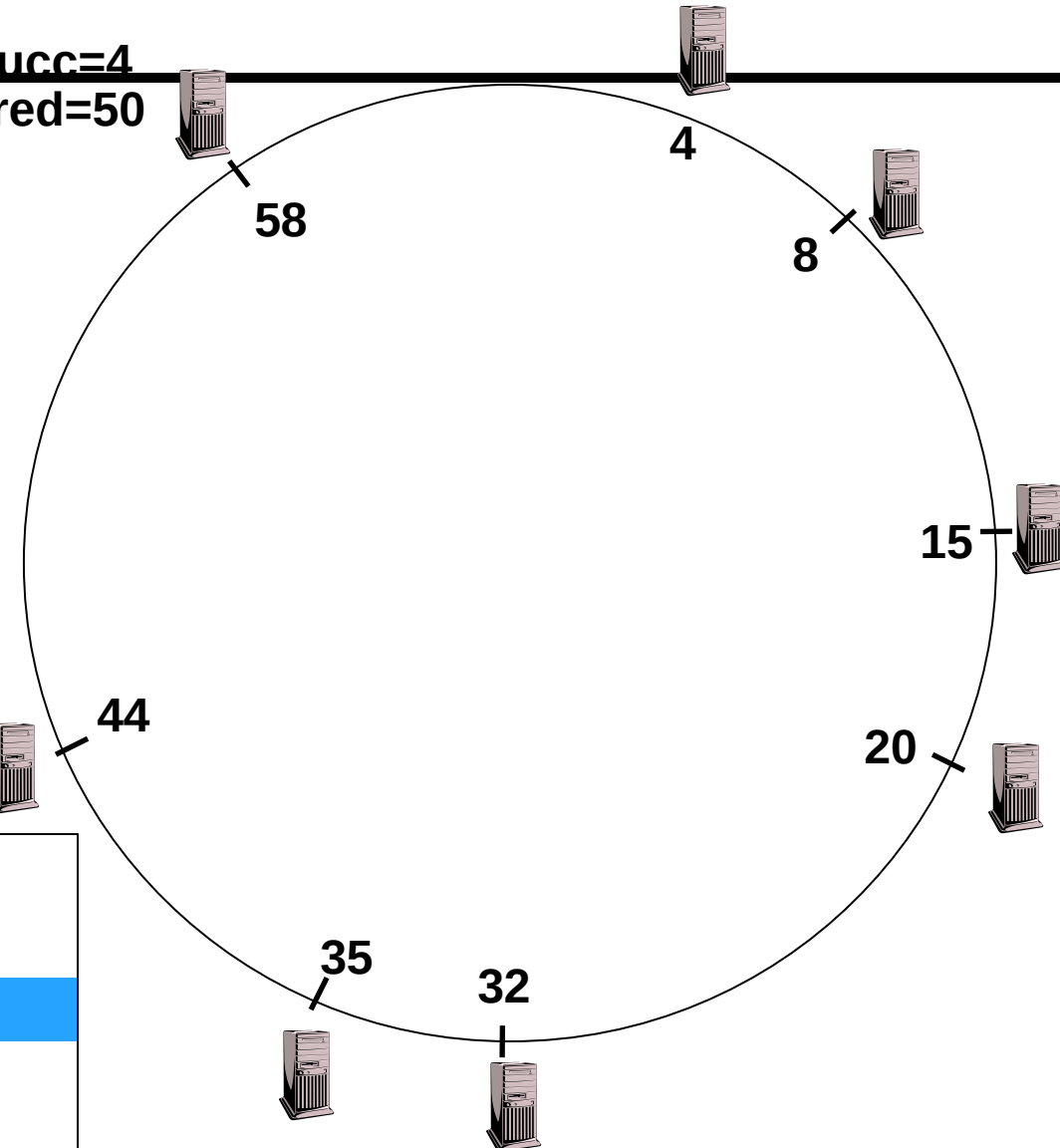
notify(44)

succ=50
pred=35

n.notify(n')

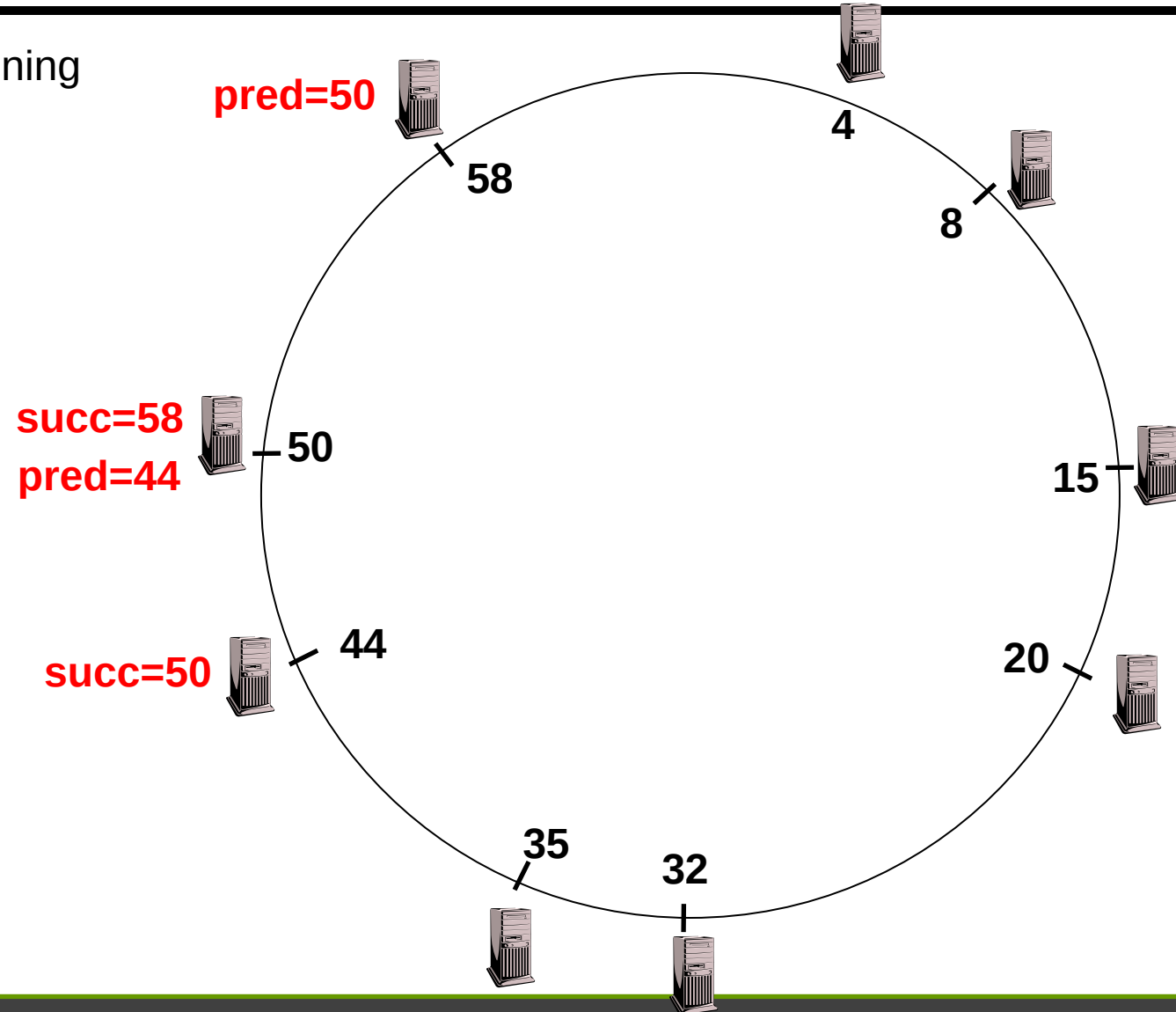
if (pred = nil or n' l(pred, n))

pred = n'



Joining Operation (cont' d)

This completes the joining operation!

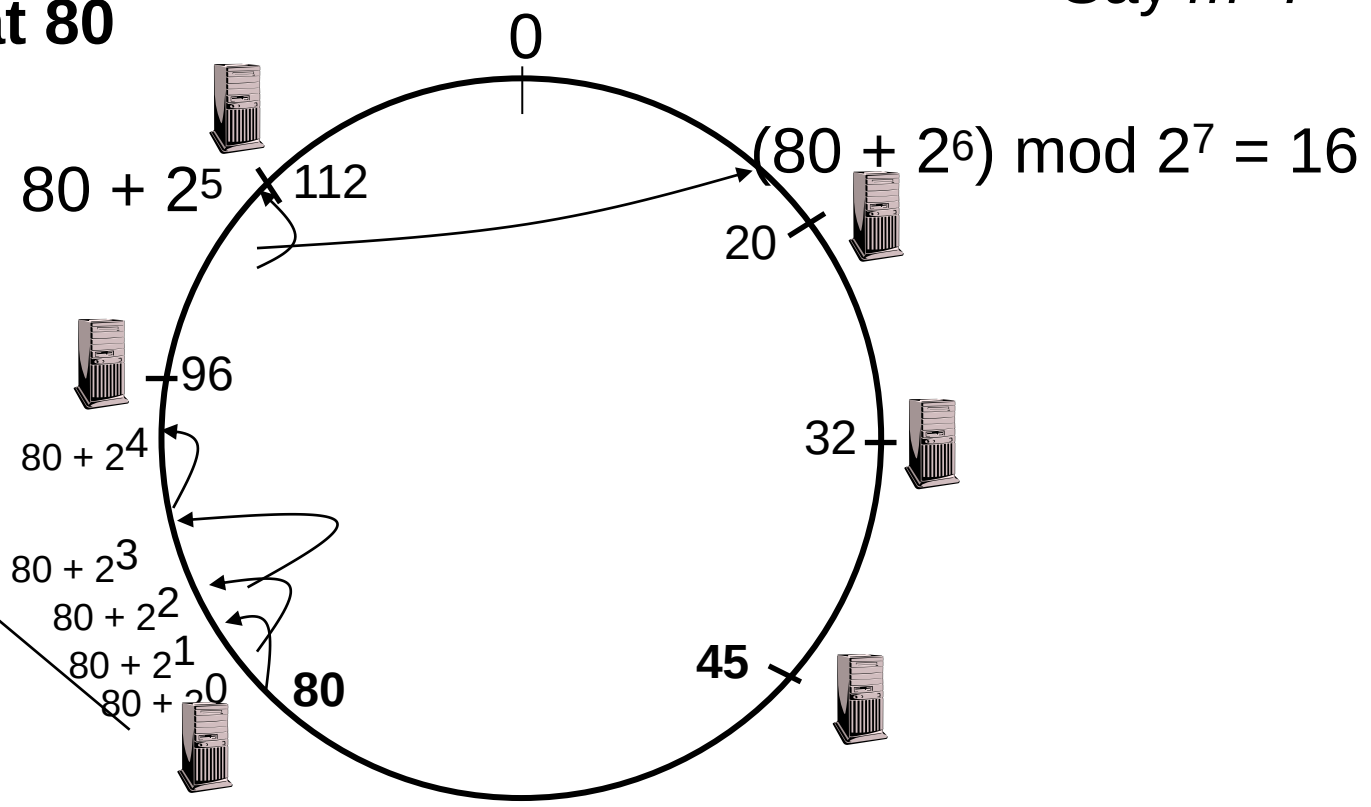


Achieving Efficiency: *finger tables*

Say $m=7$

Finger Table at 80

i	$ft[i]$
0	96
1	96
2	96
3	96
4	96
5	112
6	20



i th entry at peer with id n is first peer with id $\geq n + 2^i \pmod{2^m}$

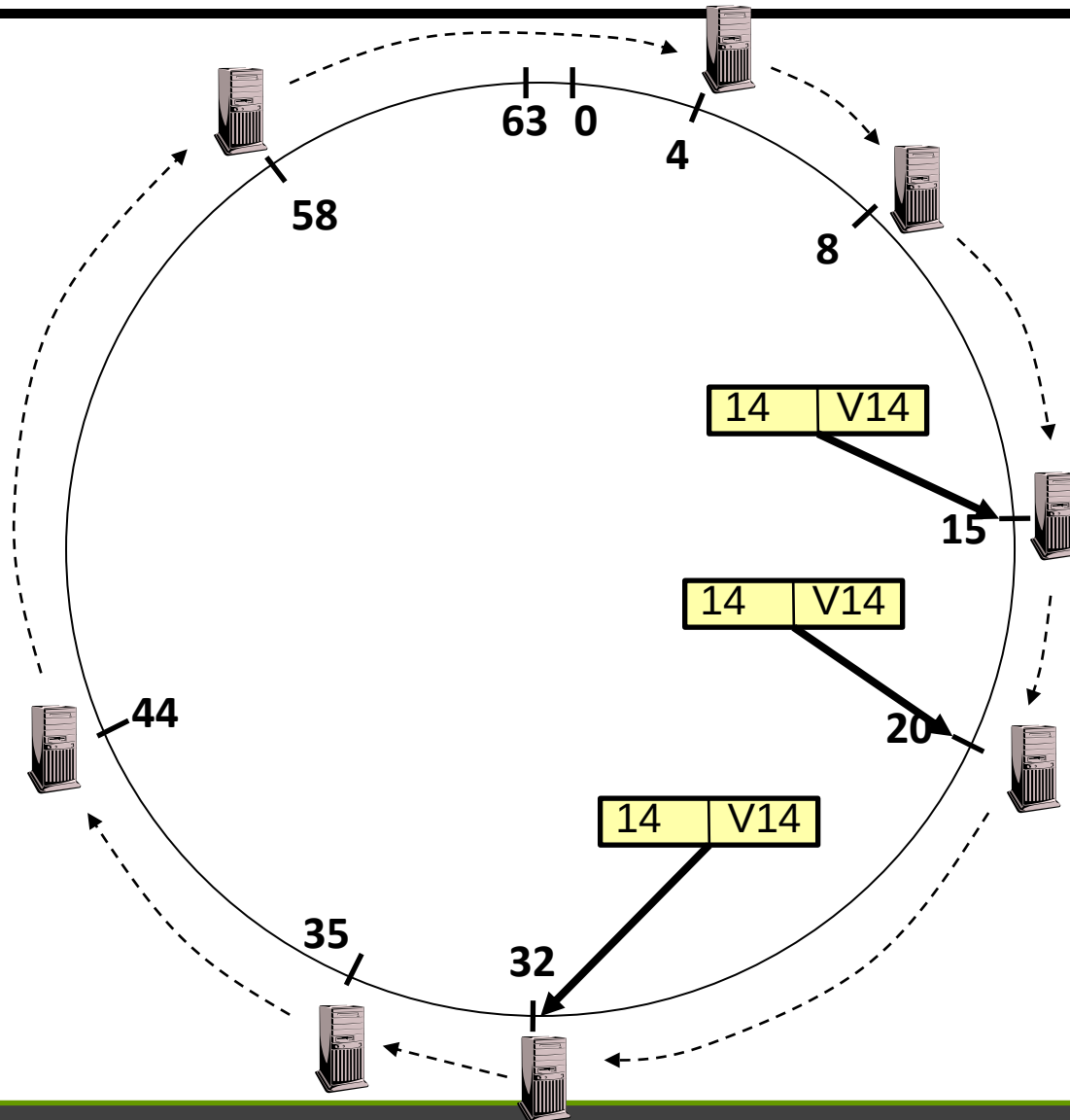
Achieving Fault Tolerance for Lookup Service

- ❑ To improve robustness each node maintains the k (> 1) immediate successors instead of only one successor
- ❑ In the **pred()** reply message, node A can send its $k-1$ successors to its predecessor B
- ❑ Upon receiving **pred()** message, B can update its successor list by concatenating the successor list received from A with its own list
- ❑ If $k = \log(M)$, lookup operation works with high probability even if half of nodes fail, where M is number of nodes in the system



Storage Fault Tolerance

- Replicate tuples on successor nodes
- Example:
replicate (K14, V14)
on nodes 20 and 32

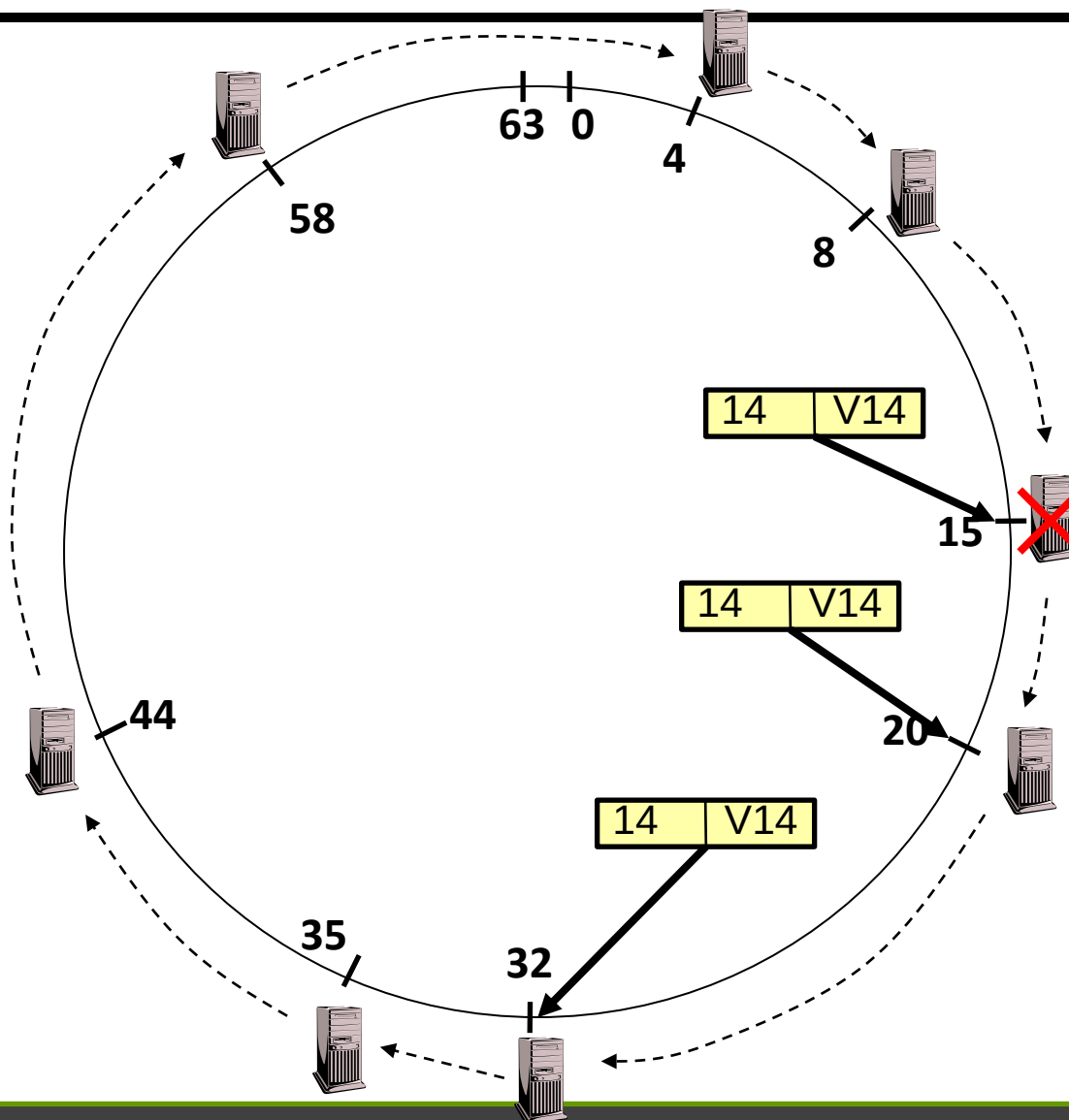


Storage Fault Tolerance

❑ If node 15 fails, no reconfiguration needed

Still have two replicas
All lookups will be correctly routed

❑ Will need to add a new replica on node 35



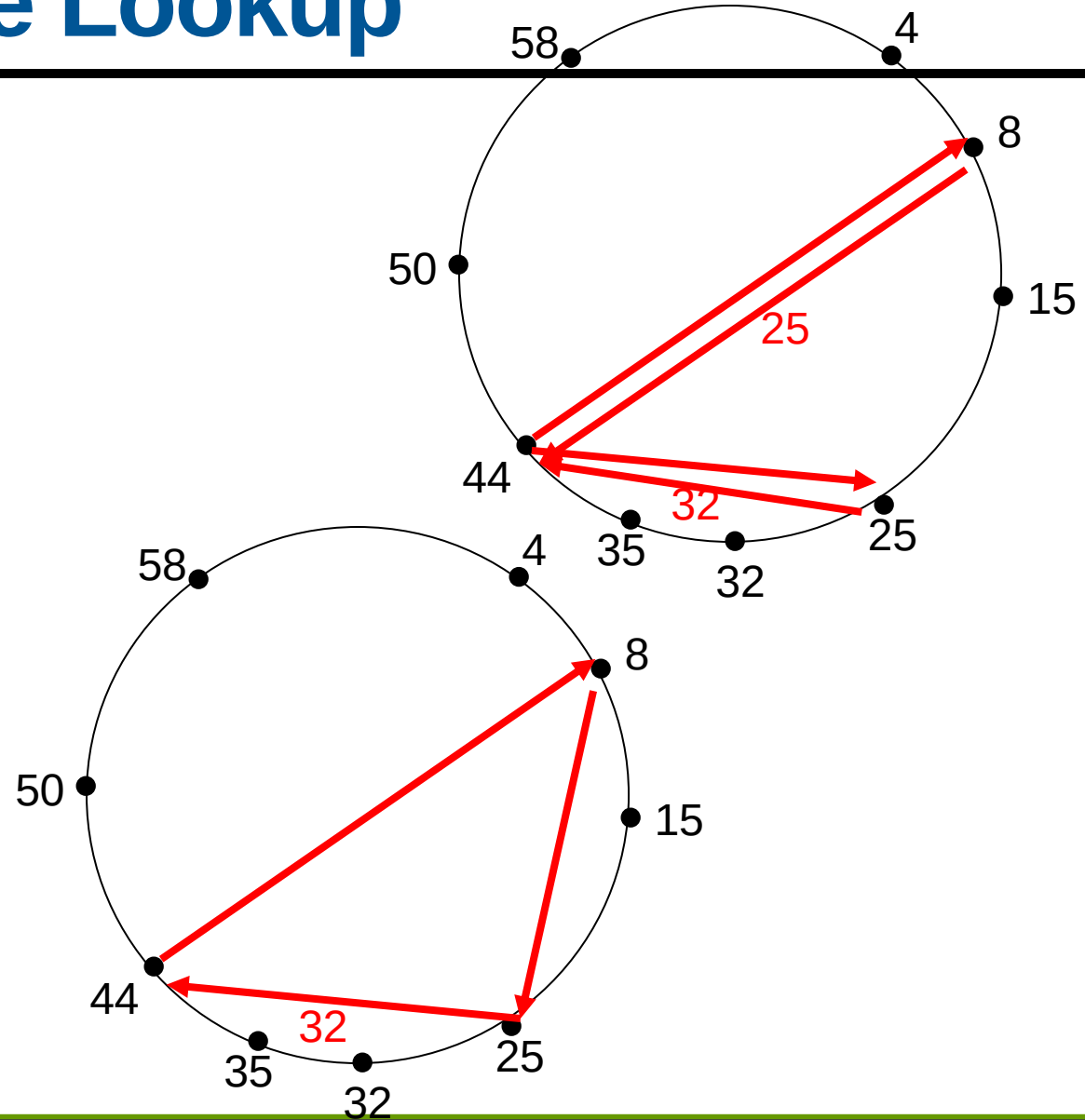
Iterative vs. Recursive Lookup

Iteratively:

- Example: node 44
issue query(31)

Recursively

- Example: node 44
issue query(31)



Others – Zookeeper, Flink, Clickhouse

□ Zookeeper

<http://zookeeper.apache.org/>

- A highly-available service for **coordinating processes** of distributed applications.

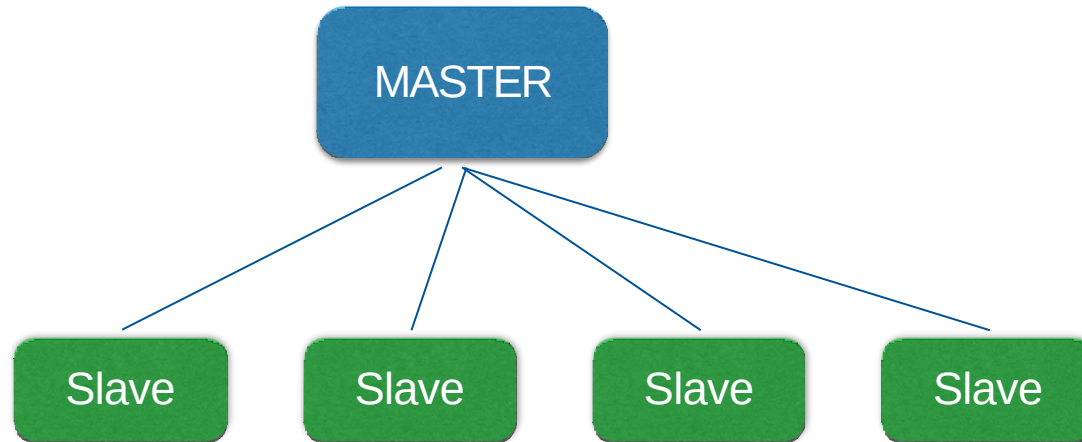
- ✓ Developed at Yahoo! Research
- ✓ Started as sub-project of Hadoop, now a top-level Apache project

■ Motivation

- In the past: a **single** program running on a **single** computer with a **single** CPU
- Today: applications consist of **independent** programs running on a **changing** set of computers
- Difficulty: **coordination** of those independent programs
- Developers have to deal with **coordination logic** and **application logic** at the same time
 - ✓ ZooKeeper: designed to relieve developers from writing coordination logic code

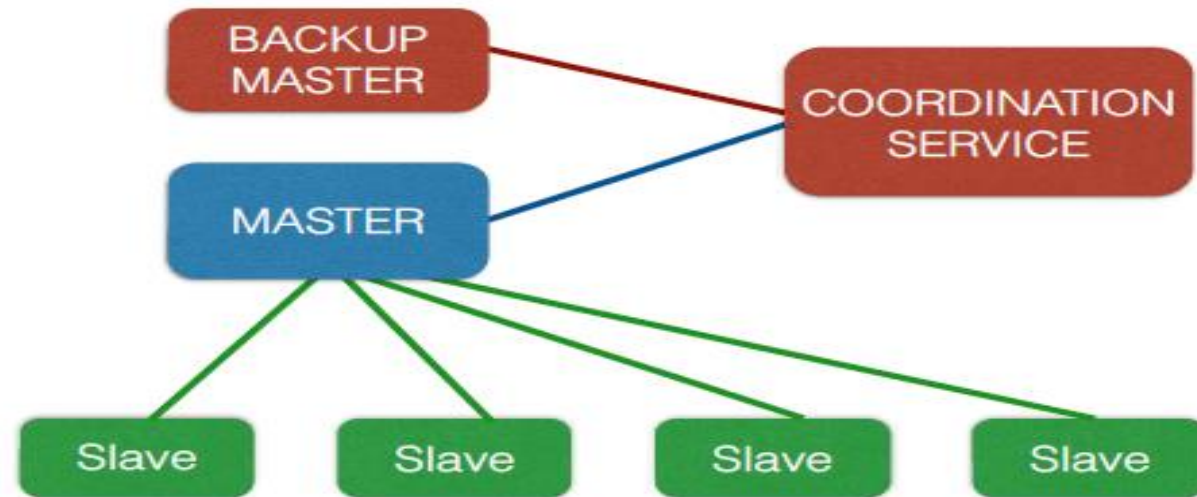


How can a distributed system look like?



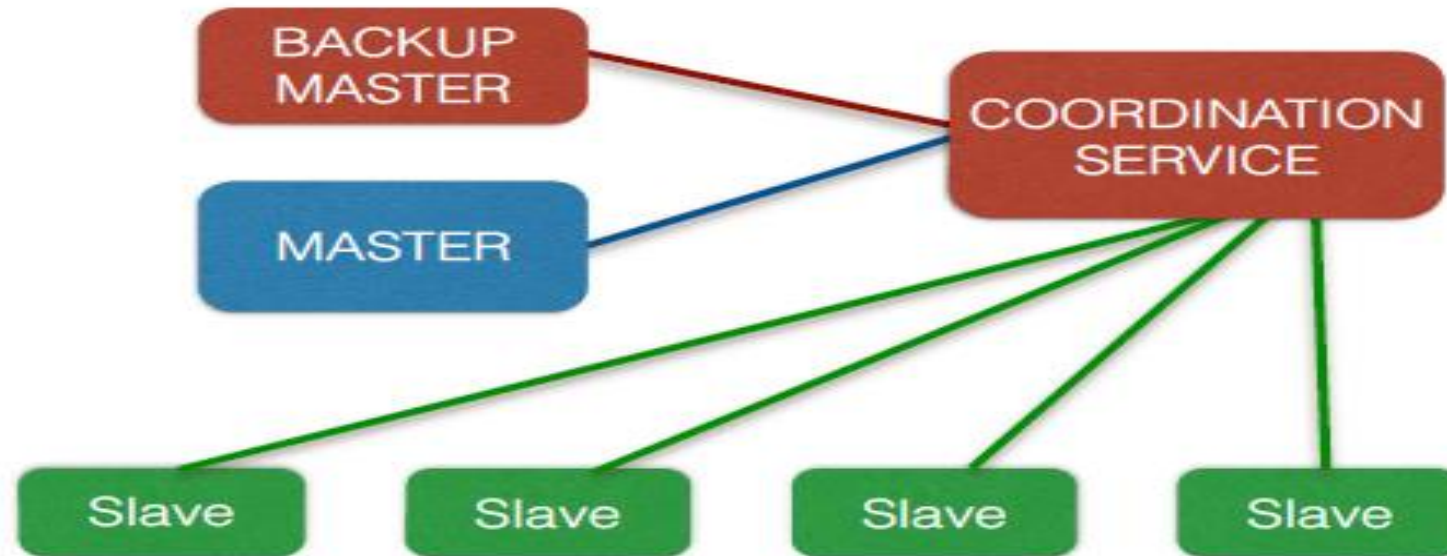
- + **simple**
- **coordination performed by the master**
- **single point of failure**
- **scalability**

How can a distributed system look like?



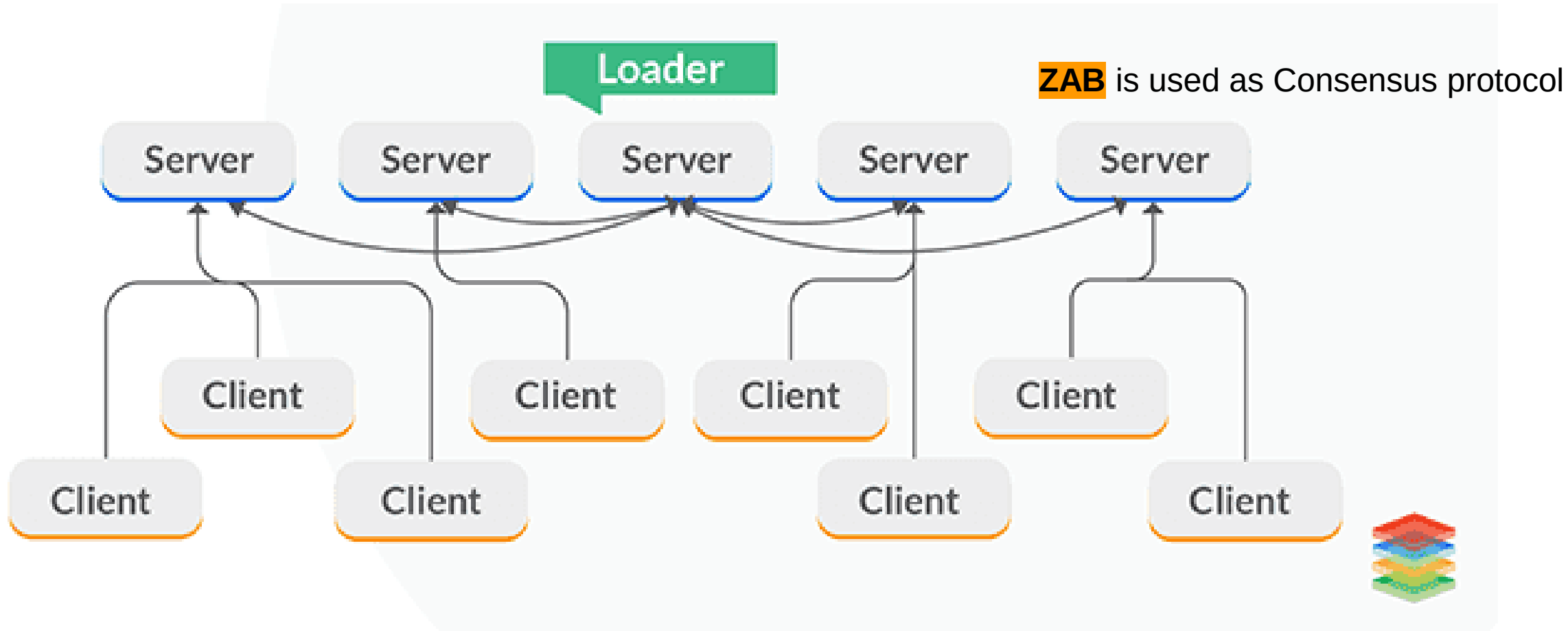
- + not a single point of failure anymore
- scalability is still an issue

How can a distributed system look like?



+ scalability

Zookeeper's architecture – a cluster of nodes



ZooKeeper terminology

- ❑ Client: user of the ZooKeeper service
- ❑ Server: process providing the ZooKeeper service
- ❑ znode: in-memory data node in ZooKeeper, organised in a hierarchical namespace (the data tree)
- ❑ Update/write: any operation which modifies the state of the data tree
- ❑ Clients establish a session when connecting to ZooKeeper

- ❑ **connect** – connect to the ZooKeeper ensemble
- ❑ **create** – create a znode
- ❑ **exists** – check whether a znode exists and its information
- ❑ **getData** – get data from a particular znode
- ❑ **setData** – set data in a particular znode
- ❑ **getChildren** – get all sub-nodes available in a particular znode
- ❑ **delete** – delete a particular znode and all its children
- ❑ **close** – close a connection

```
create(String path, byte[] data, List<ACL> acl, CreateMode mode)
```

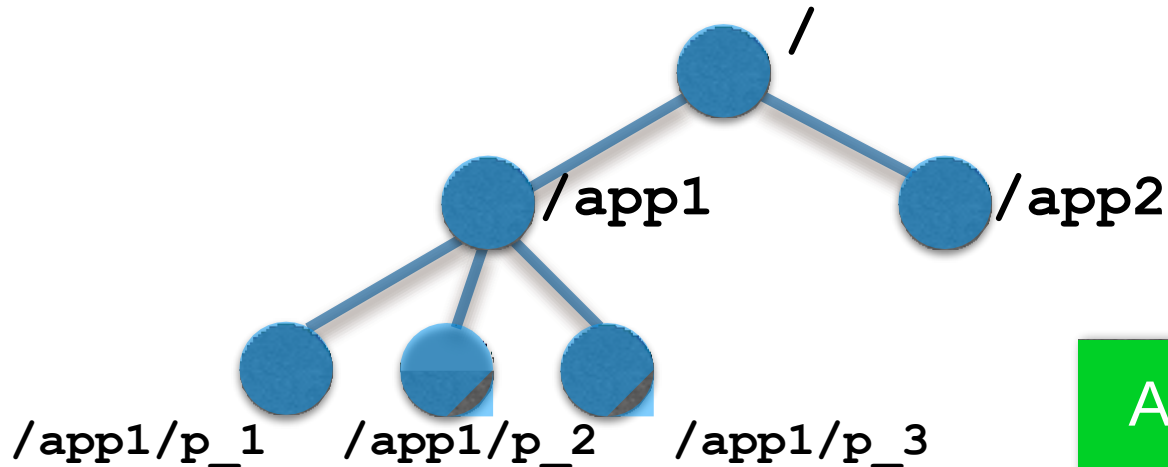
```
exists(String path, boolean watcher)
```

```
getData(String path, Watcher watcher, Stat stat)
```



ZooKeeper's data model: filesystem

- ❑ znodes are organised in a hierarchical namespace
- ❑ znodes can be manipulated by clients through the ZooKeeper API
- ❑ znodes are referred to by UNIX style file system

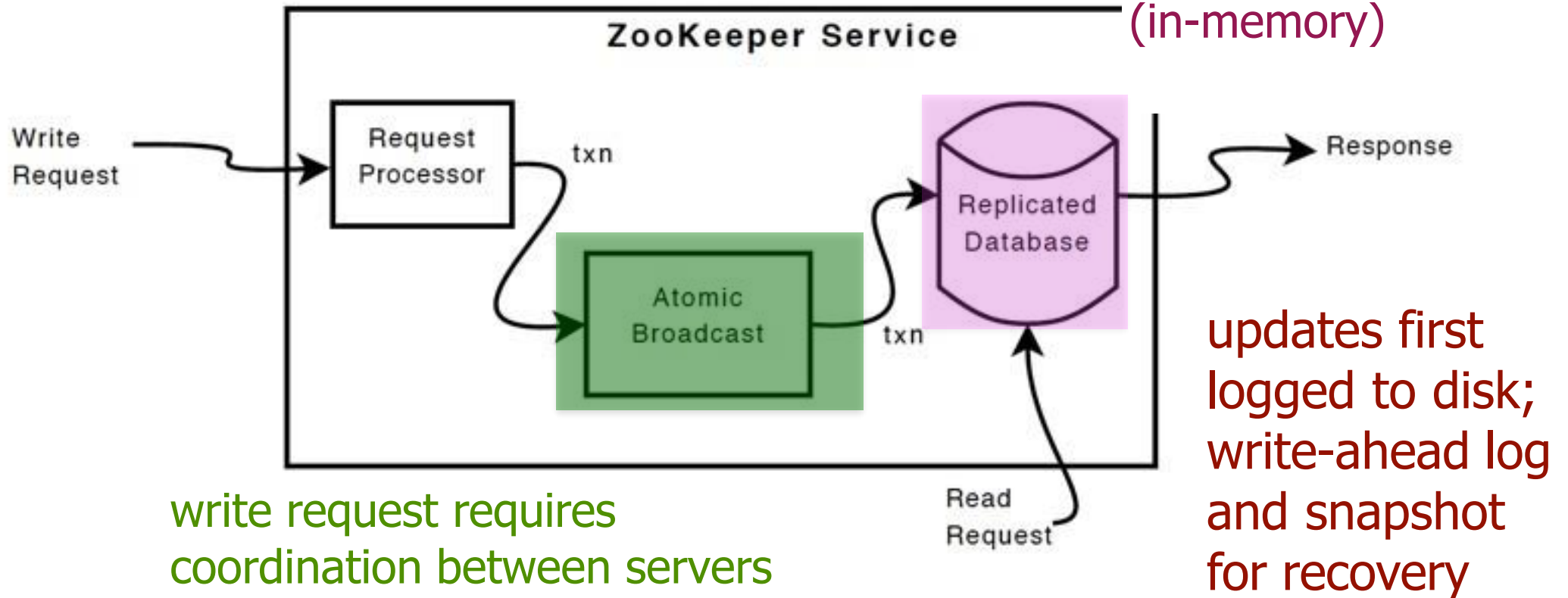


All znodes store data **(file like)** & can have children **(directory like)**.

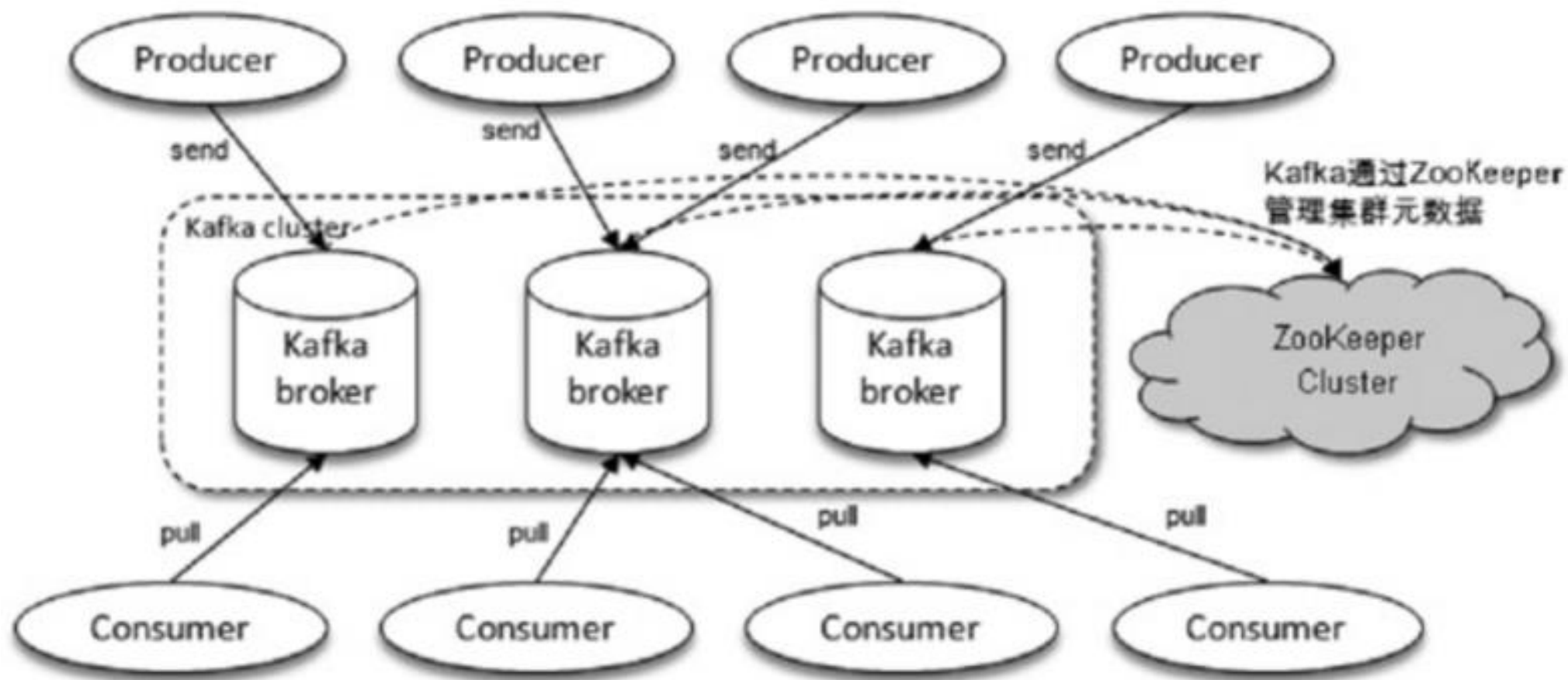
A few implementation details

- ZooKeeper data is replicated on each server that composes the service

replicated across
all servers
(in-memory)



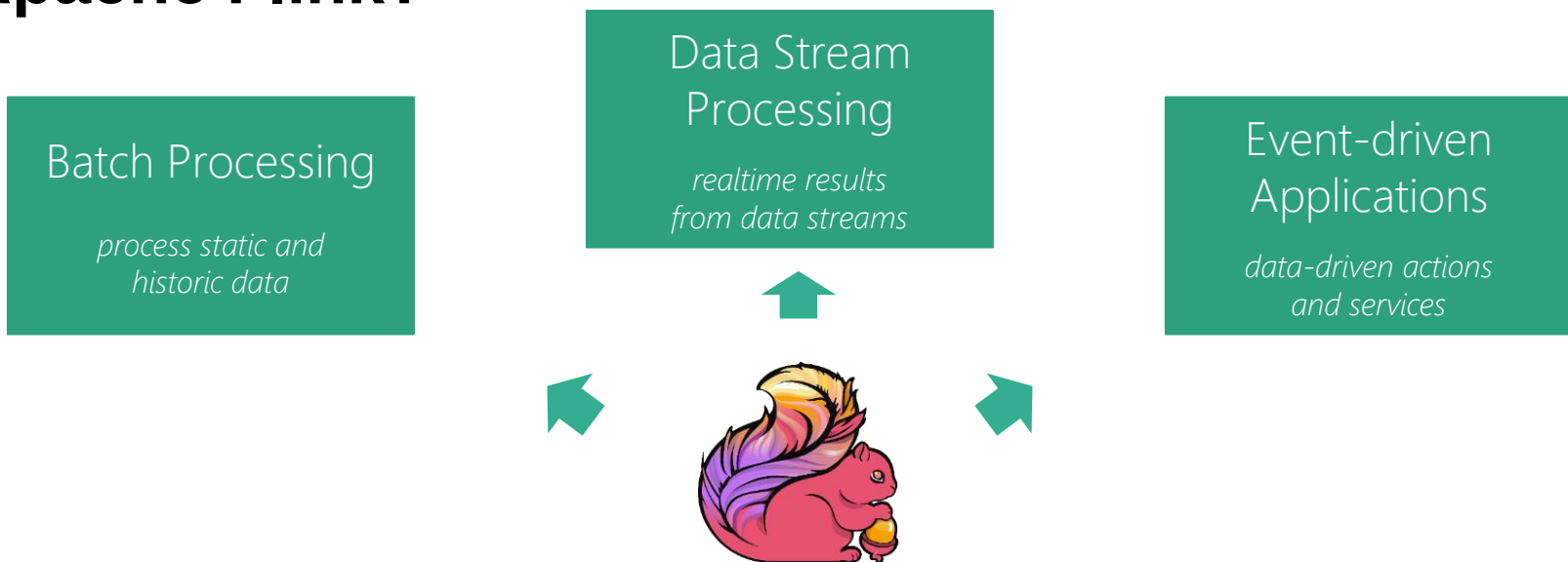
生产者将消息发送到broker



消费者采用拉 (pull) 模式订阅并消费消息

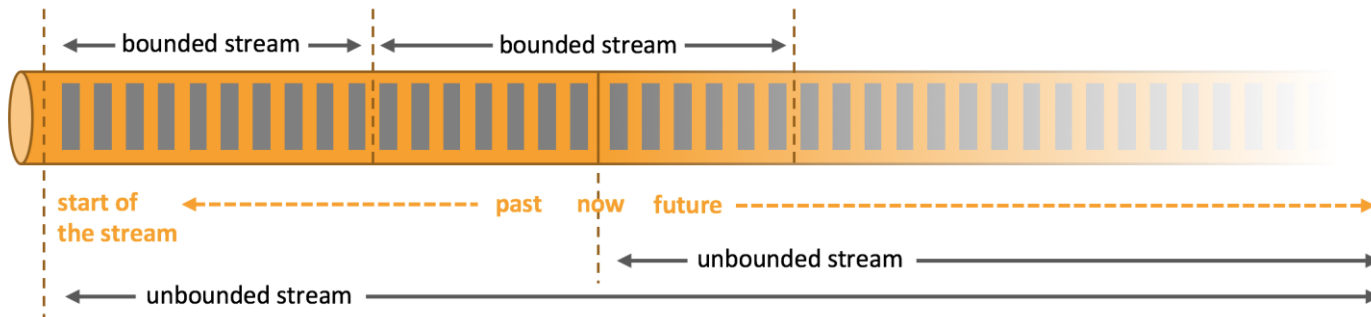


What is Apache Flink?



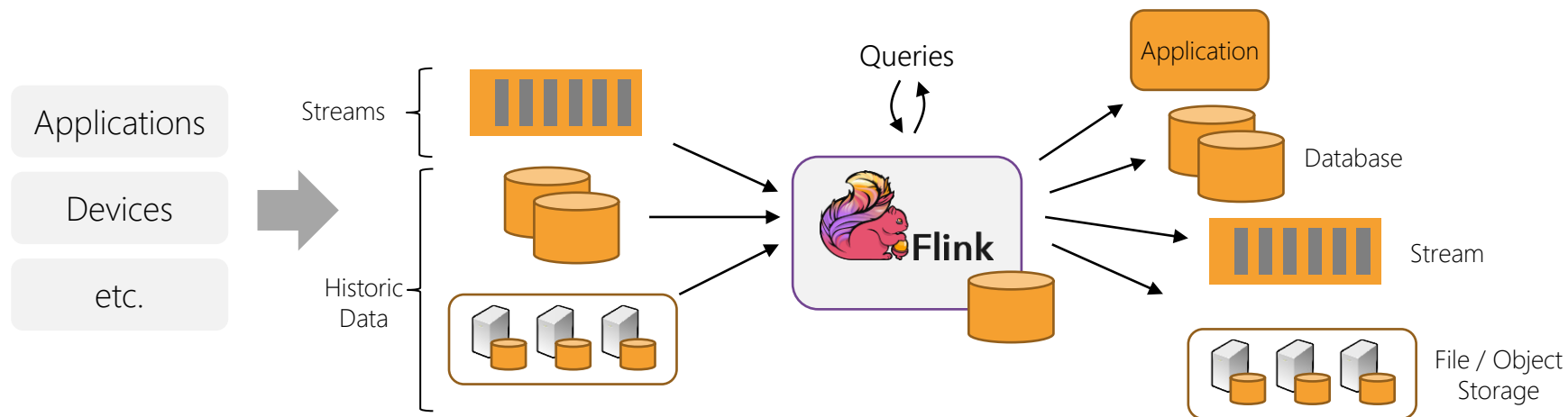
Stateful Computations Over Data Streams

Everything Streams



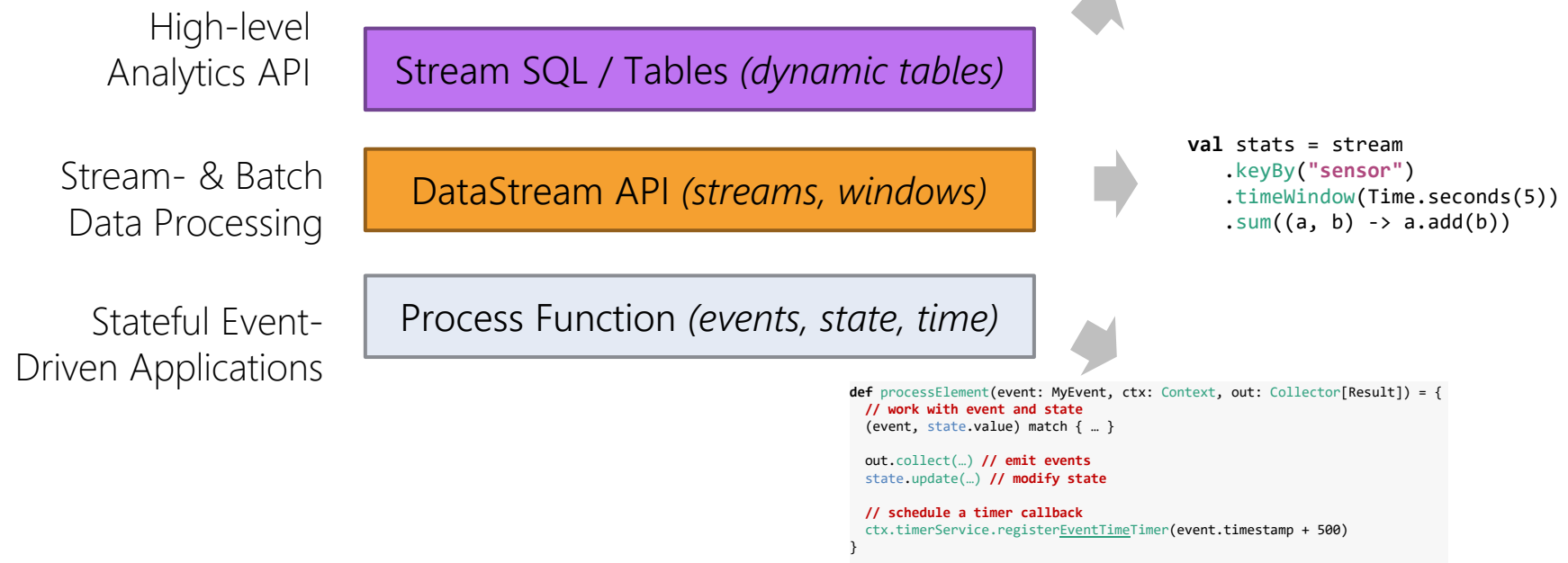
□ Apache Flink in a Nutshell

Stateful computations over streams
real-time and historic
fast, scalable, fault tolerant, in-memory,
event time, large state, exactly-once

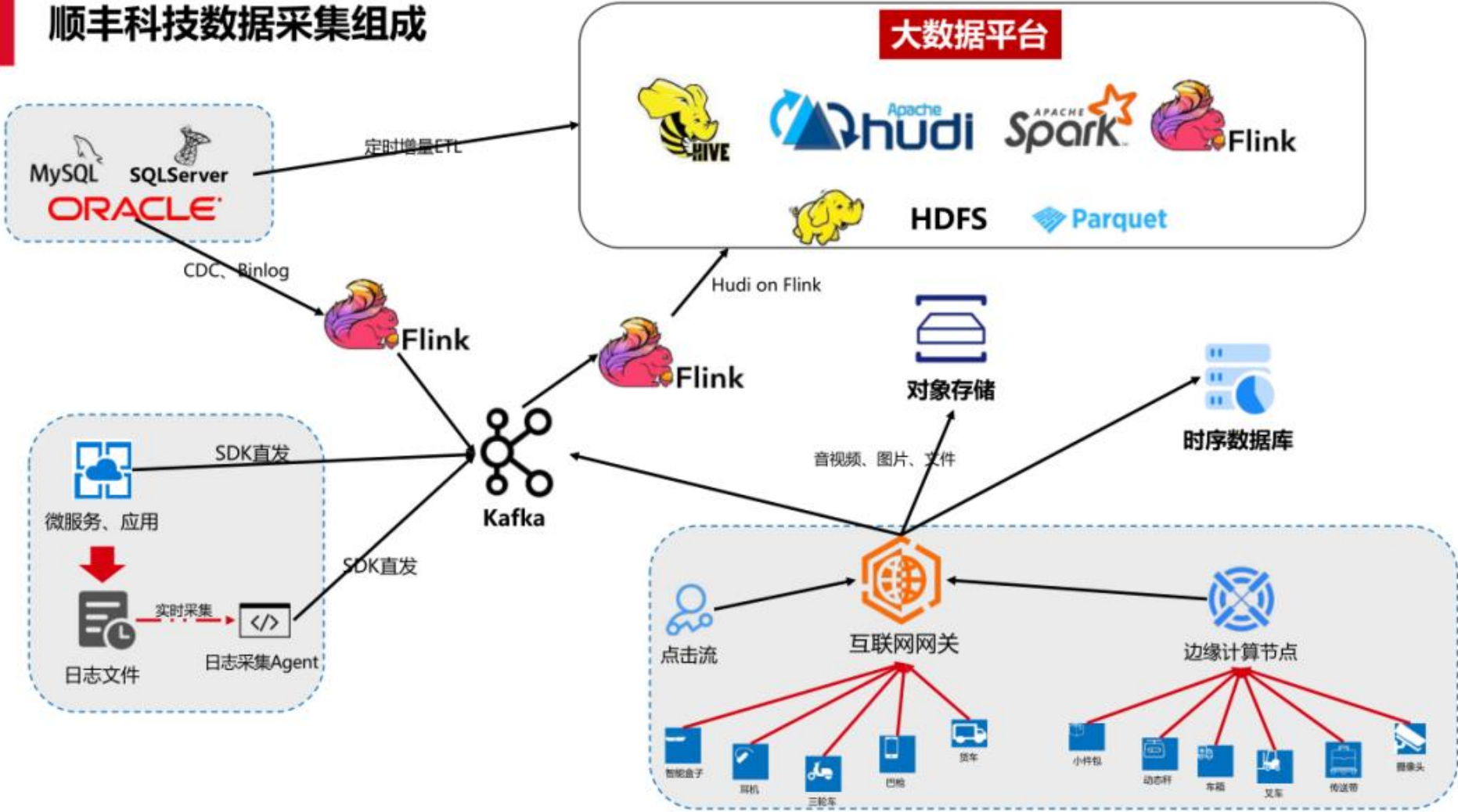


□ Powerful Abstractions

Layered abstractions to
navigate simple to complex use cases



顺丰科技数据采集组成



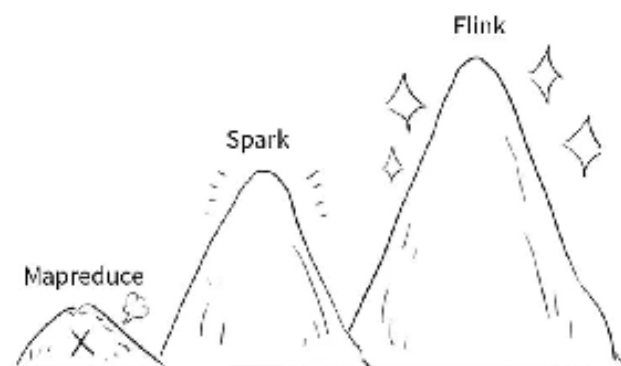


Flink

Apache Flink 是一个分布式大数据处理引擎，可对有限数据流和无限数据流进行有状态计算。可部署在各种集群环境，对各种大小的数据规模进行快速计算。

Realtime Compute

阿里云实时计算（Alibaba Cloud Realtime Compute）则是一套基于Apache Flink构建的一站式、高性能实时大数据处理平台，广泛适用于流式数据处理、离线数据处理等场景。



一眨眼，所有东西都计算好了！



Eron Wright
@eronwright

<https://raw.githubusercontent.com/flink-china/flink-forward-sf-2017/master/slides/Introducing%20Flink%20Tensorflow.pptx>

TensorFlow & Apache Flink™

An early look at a community project

<https://github.com/cookieai/flink-tensorflow>

Apache®, Apache Flink™, Flink™, and the Apache feather logo are trademarks of [The Apache Software Foundation](https://www.apache.org/).
TensorFlow, the TensorFlow logo and any related marks are trademarks of Google Inc.



clickhouse是什么

ClickHouse是一个面向联机分析处理(OLAP)的开源的面向列式数据库管理系统(DBMS)。

在 2016 年开源，开发语言为C++，是一款 PB 级的交互式分析数据库。

与Hadoop, Spark相比，ClickHouse很轻量级。

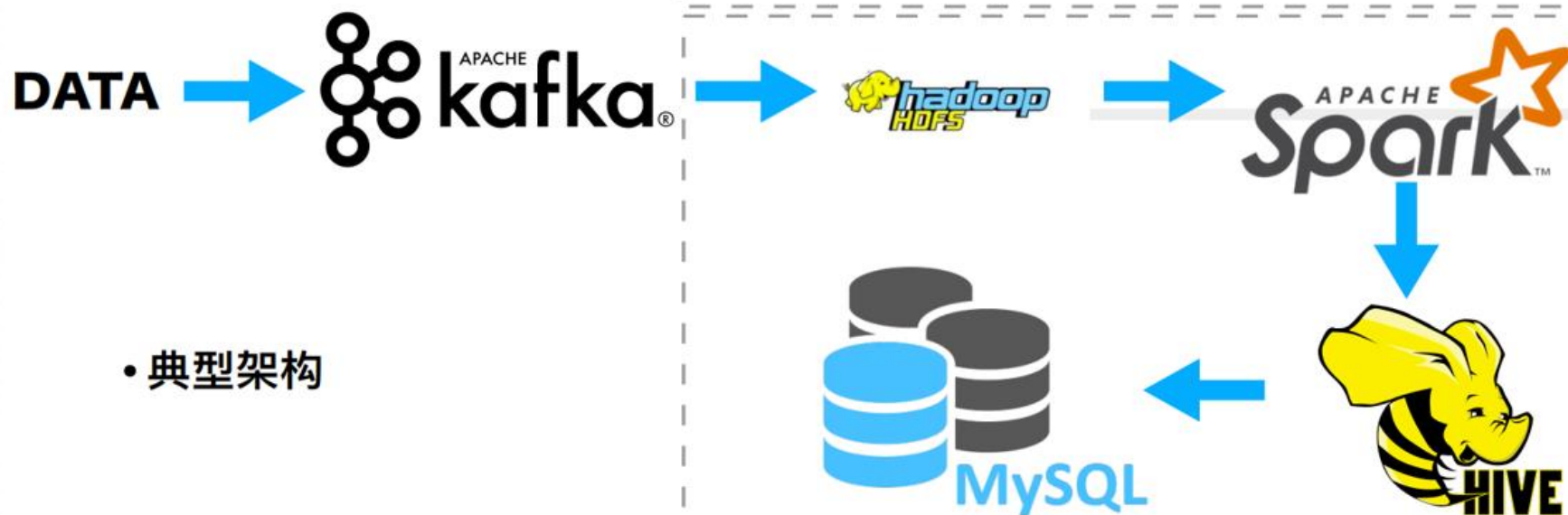


https://blog.csdn.net/weixin_39025362/article/details/114114191

传统链路

Why ClickHouse

Before ClickHouse



• 典型架构

CK链路



新与旧的对比

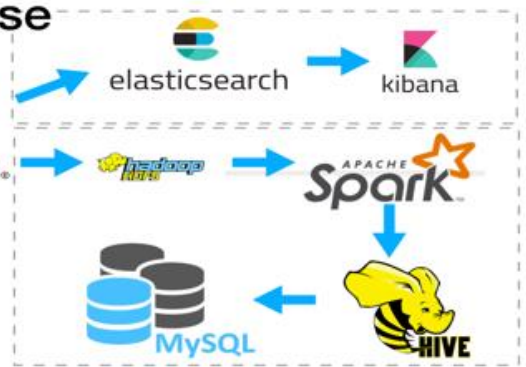


Why ClickHouse

Before ClickHouse



• 典型架构



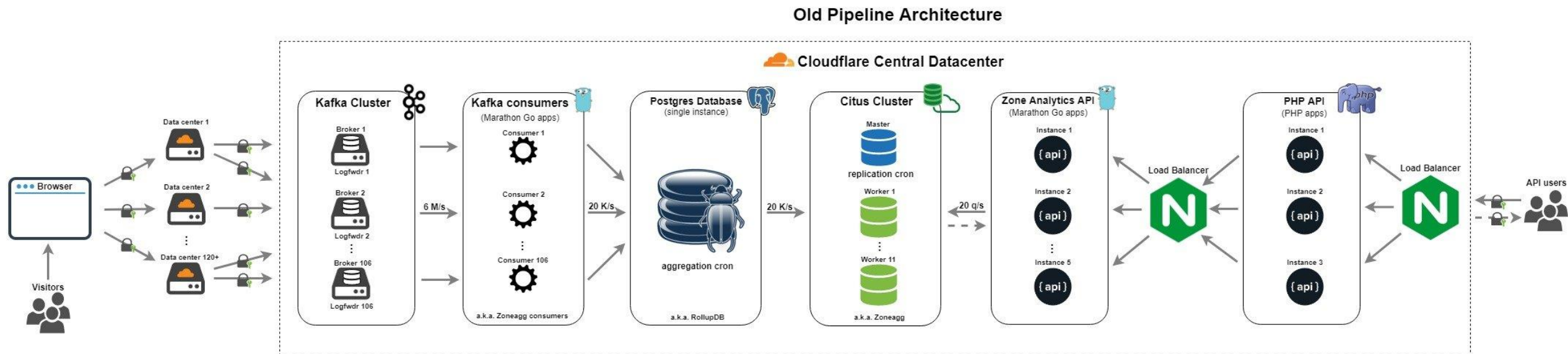
- 1 链路短，瓶颈少
- 2 组件少，维护成本低
- 3 类sql查询，上手快
- 4 占用资源少

https://blog.csdn.net/weixin_39025362

https://blog.csdn.net/weixin_39025362/article/details/114114191

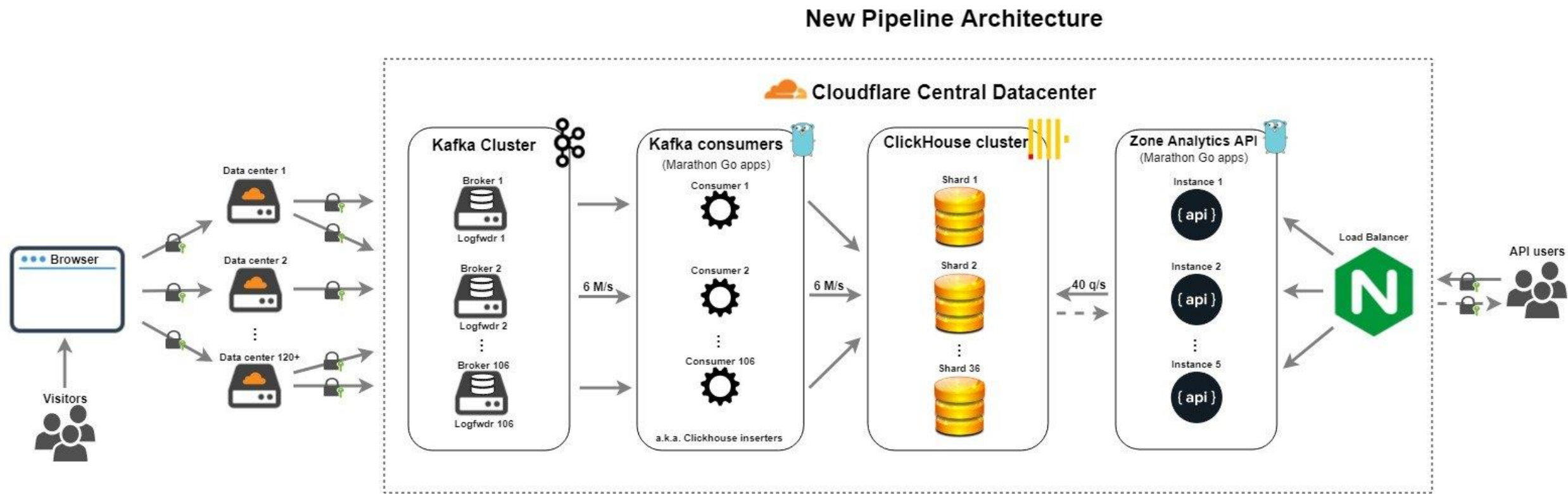
□ 使用 ClickHouse 来分析处理每秒 600 万 HTTP 请求

■ 之前的数据管道



□ 使用 ClickHouse 来分析处理每秒 600 万 HTTP 请求

■ 全新的数据管道



我们的 ClickHouse 集群

我们总共有 36 个 ClickHouse 节点。新的硬件对我们来说是一个很大的提升:

- **Chassis** - 广达 D51B-2U 机箱升级为广达 D51PH-1ULH 机箱 (减少2倍的物理空间)
- **CPU** - 32 个核心的 E5-2630 v4 @ 2.20 GHz 升级为 40 个逻辑核心的 E5-2630 v3 @ 2.40 GHz
- **内存** - 128 GB 内存升级到 256 GB 内存
- **硬盘** - 12 x 6 TB 东芝 TOSHIBA MG04ACA600E 升级为 12 x 10 TB 希捷 ST10000NM0016-1TT101
- **网络** - 2 x 10G Intel 82599ES 升级为 2 x 25G Mellanox ConnectX-4 in MC-LAG

我们的平台运营团队发现, ClickHouse 还不擅长运行异构集群, 所以我们需要逐步将现有集群中的全部 36 节点更换为全新的硬件。这个过程相当简单, 跟更换故障节点没什么区别。问题是 ClickHouse 无法 throttle recovery。

clickhouse社区

数据库	开源时间	Github star 数量
clickhouse	2016年	14.1k
kylm	2015年	2.9k
impala	2012年	2.3k

ClickHouse / ClickHouse

Watch 609 Star 14.1k Fork 2.8k

<> Code Issues 1.4k Pull requests 141 Actions Projects Wiki Security Insights

Pulse

Contributors

Community

Commits

Code frequency

Dependency graph

Network

Forks

Feb 22, 2009 – Jan 10, 2021

Contributions: Commits

Contributions to master, excluding merge commits



目前国内社区火热，各个大厂纷纷跟进大规模使用：

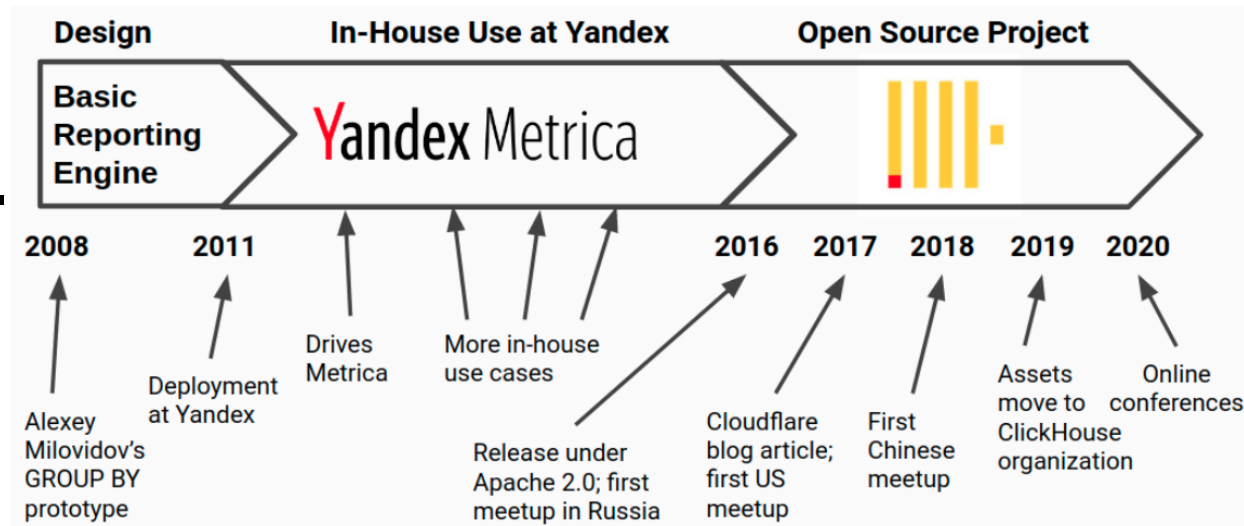
ByteDance
字节跳动

Tencent 腾讯

Ctrip
携程

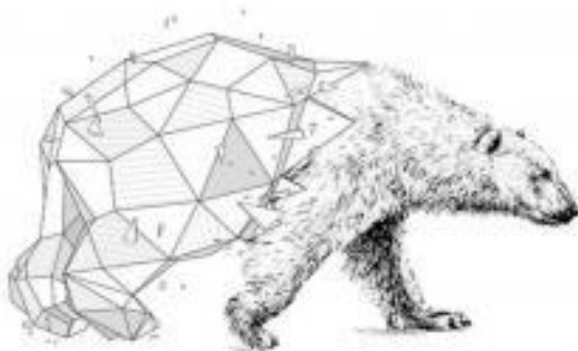
https://blog.csdn.net/weixin_39025362

https://blog.csdn.net/weixin_39025362/article/details/114114191



ClickHouse开发团队负责人及核心贡献者亲作序推荐，ClickHouse华人社区与大数据领域多位专家联袂推荐
ClickHouse贡献者和读者最佳选择，从核心理念、基础功能、运行原理以及应用等多个维度，对ClickHouse
进行全方位解析。

数据库
技术丛书



ClickHouse 原理解析与应用实践

ClickHouse Principle and Practice

朱凯 著

机械工业出版社
China Machine Press

□ ClickHouse原理解析与应用实践

□ 朱凯

□ 2020

□ 北京华章图文信息有限公司

ClickHouse就是这样一款拥有卓越性能的OLAP数据库，是目前业界公认的OLAP数据库黑马，有很大的发展潜力，并且已经在许多企业的内部得到应用。

一次机缘巧合，在研究BI产品技术选型的时候，我接触到了ClickHouse，瞬间就被其惊人的性能所折服。这款非Hadoop生态、简单、自成一体的技术组件引起了我极大的好奇。那么ClickHouse究

