GemFire与12306

GemFire介绍

DTCC

2015中国数据库技术大会

DATABASE TECHNOLOGY CONFERENCE CHINA 2015

大数据技术探索和价值发现

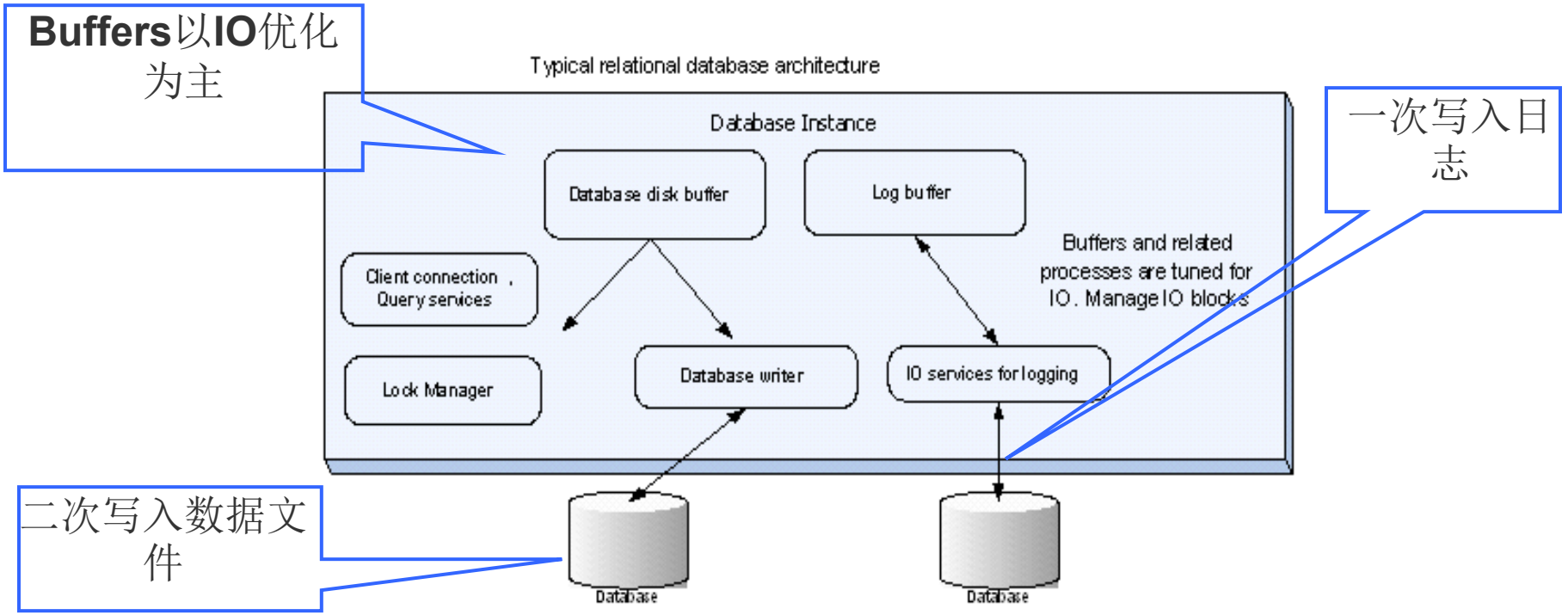
IT168 ChinaUnix ITPUB

有状态架构所面临的挑战

- **应用服务器在高负载的压力下可以很好地进行扩展，但是应用状态管理还必须与数据库服务器进行交互**

- **关系型数据库设计植根于过去的技术**
  - 固有的I/O瓶颈，强一致性锁机制
  - 集中架构设计导致扩展瓶颈
- **云环境是虚拟化的、分布式的**
  - 虚拟化的分布式架构并不适合RDBMS
- **高可用性有限**
  - 完全双活的高可用性难以实现

传统关系型数据库所面临的挑战



- 频繁的I/O读写
- 设计架构不适应现在的应用特点
  - 对 ACID 关注太多
  - 磁盘同步瓶颈

GemFire的前世今生

1982年

GemStone Systems成立于美国俄勒冈州西北部城市比弗顿市。GemStone是领先的分布式数据管理技术软件公司，在此方面有着厚重的历史。最初使用Smalltalk语言开发出了第一代面向对象的数据库。并成为Smalltalk执行委员会成员。

1986年

第一代产品Gemstone/S 正式面世，受到金融市场的广泛欢迎。

90年代中后期

随着Java语言的广泛应用，GemStone与Sun公司合作，参与到JEE的规范制定(JCache -JSR107)，并陆续更新与JEE平台相结合的产品。GemStone开发出了GemFire，成为业界第一个满足J2EE标准的中间件。GemFire拥有全新的应用框架，兼容Java, C++, C#。而GemFire在CEP(complex event processing)，Event Stream Processing，Data Virtualization， Distributed Caching几个方面有着举足轻重的地位。

2008年初

金融危机后，金融监管法规Dodd-Frank、Basel3等陆续出台，各大投资银行为了减少系统风险和增加透明度，加强了金融衍生品交易平台的投资规模，Gemfire击败Oracle等老牌厂商，跻身为华尔街第一大分布式数据处理平台软件。

2010年5月

VMware收购了老牌厂商Gemstone，并入SpringSource部门。

什么是GemFire？

基于内存的海量数据实时处理平台

分布式池化内存和硬盘资源

- 超低延迟和超高吞吐量
- 动态的线性扩展能力
- 持续高可用性
- **MapReduce**并行处理
- 数据感知路由

弹性架构扩展系统

A seminal paper on the architecture of elastic applications was written by **Pat Helland** (Tandem Computing, Amazon.com, Microsoft)

"Life Beyond Distributed Transactions: an Apostate's Opinion"

http://www.cidrdb.org/cidr2007/papers/cidr07p15.pdf

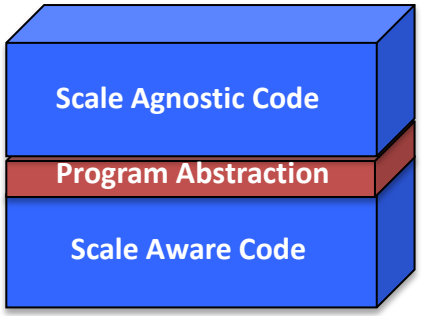
http://blogs.msdn.com/b/pathelland/

这片论文描述了我们的应用系统架构需要怎样的步骤
来达到线性扩展,同时不使用大型硬件产品

面向分层架构

Helland 指出了线性扩展应用的分层架构

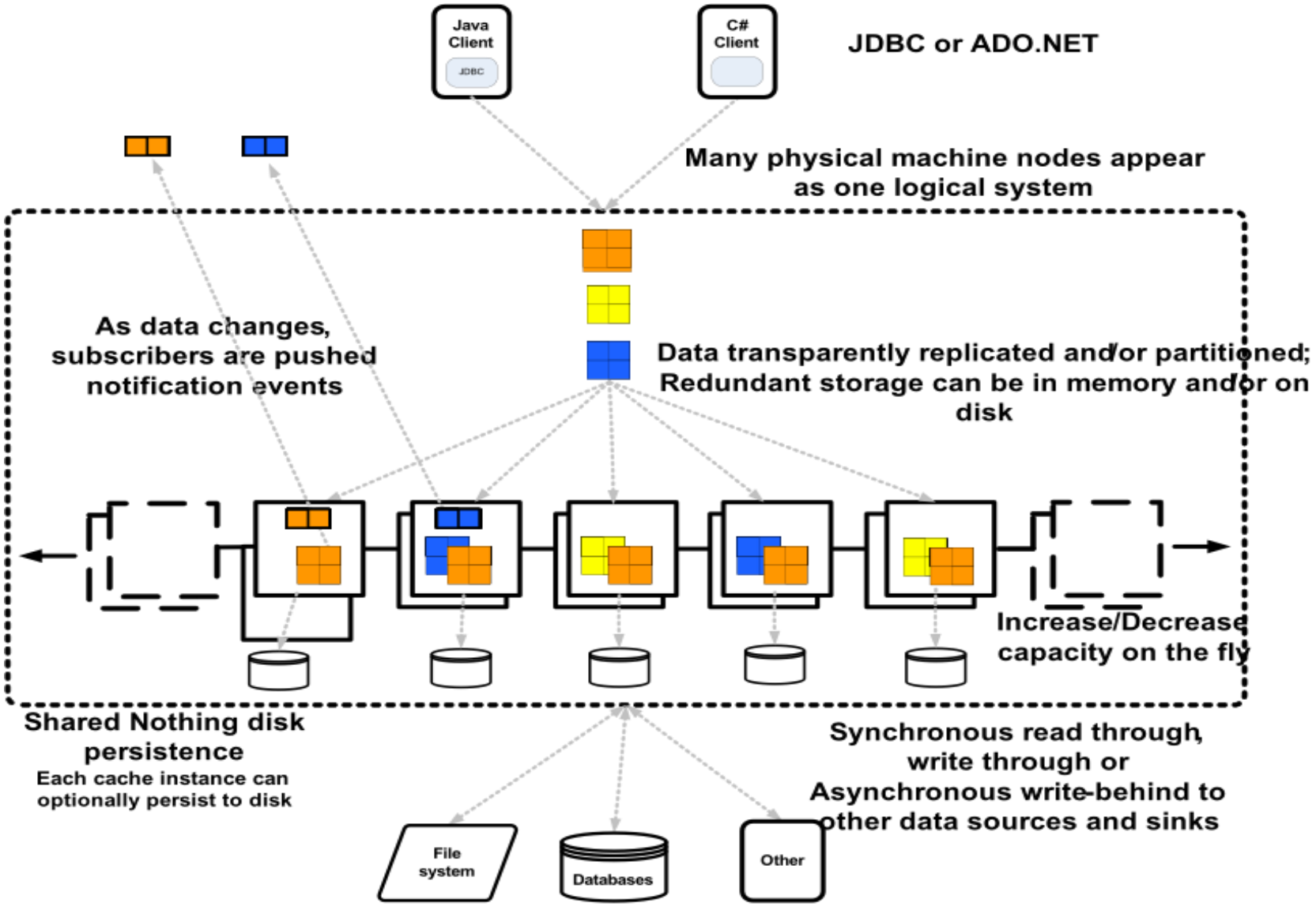
顶层不必担心扩展问题



中间抽象层

底层知道应用是跨多台物理机分布的

GemFire整体分布式架构



GemFire基本概念介绍

Gemfire Distributed System	运行在Gemfire上的VMs组成了分布式系统。每一个MV都作为一个Gemfire对等体存在。启动Gemfire对等体，在每个VM对等体上你创建了一个缓存。每一个缓存管理到其他VM对等体的连接。通过UDP多播或者TCP位置服务，互相发现。
Regions	Region是一个分布式系统之上的抽象概念。一个Region允许你在系统的多个VM中存储数据，不用考虑数据存在那个对等体上。Region给你一个map接口能透明地从合适的VM上获取数据。这个Region类扩展了java.util.Map接口，但是它也支持查询和事务。
Replicated Regions	一个Replicated Region保存着所有分区的数据拷贝
Partitioned Regions	Partitioned Regions只保存一部分分区的数据拷贝
Dynamic Regions	分布式成员，如p2p,
Client Caching	Gemfire分布式系统是一个网格结构，所有的Peer直接互相连接。Gemfire也支持客户端，客户端有他们自己的本地数据缓存，他们能够更新他们的本地缓存，通过注册服务器来更新改变的数据。
Shared-Nothing Persistence	Gemfire支持’非共享’持久化，每一个peer持久化数据到本地磁盘，Gemfire持久化也允许你在磁盘上维护一份配置的数据拷贝。
Distributed Member	在 Gemfire 分布式系统中，加入到集群中的某个成员，有多种角色。
Distributed Transaction	在 Gemfire 分布式系统中，在多个网络节点上访问或更新数据，跨节点来协调事务提交或回滚。
Distributed Lock	通过特定的分布式管理器定义一个专有资源，跨整个分布式系统锁定任意的名称。
Cache Client	接入到Gemfire分布式系统的客户端，负责管理本地缓存的生命周期。
Cache Server	Gemfire分布式系统的服务器节点。
Distribution Locator	注册客户端和服务器成员，使双方能够互相发现，同时可以提供一定的负载均衡功能。
CQ	Gemfire分布式系统能够为客户端提供持续查询服务，一旦服务器内的数据发生变化，可立刻同步到客户端。
Gateway	外部分布式系统的本地代理端，主要作为跨WAN网同步数据的出口节点。

GemFire核心功能层

Data Managerment Layer

数据保存,查询和为客户端提供访问接口。此层也负责跨分布式系统同步内存中的数据。

Membership and Discovery

分布式系统通过指定网络协议、地址和端口来发现系统成员。成员关系和发现机制动态更新系统成员列表，跨分布式系统维护统一的集群成员视图。

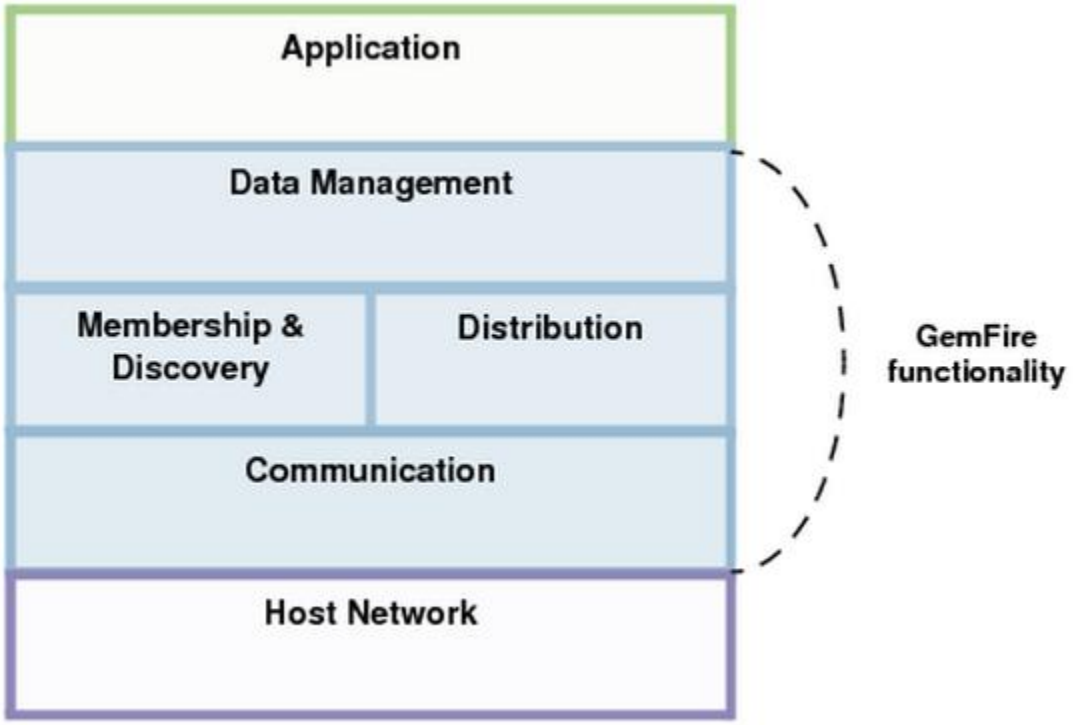
Data Distribution and Notification

数据能够跨多个主机进行同步和分区。数据更新和删除操作可接收到其他成员的ACK确认。

缓存中数据的更新、通告发送到注册了数据更新的应用上。

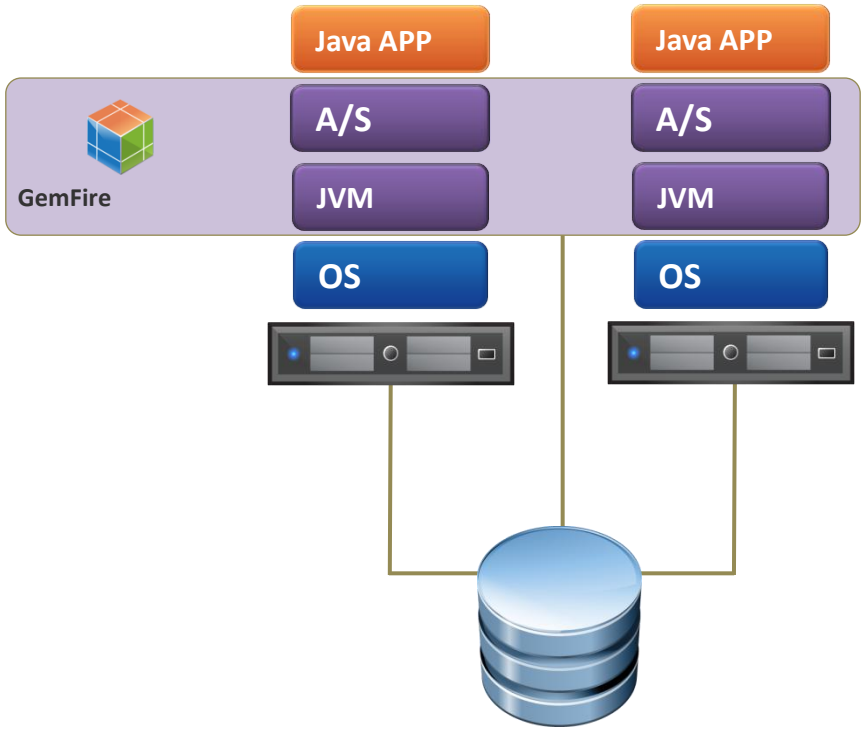
Communication

通信层使用TCP/UDP协议进行消息通信，默认情况下，发现协议是基于UDP多播机制，数据分布是基于TCP/IP协议。



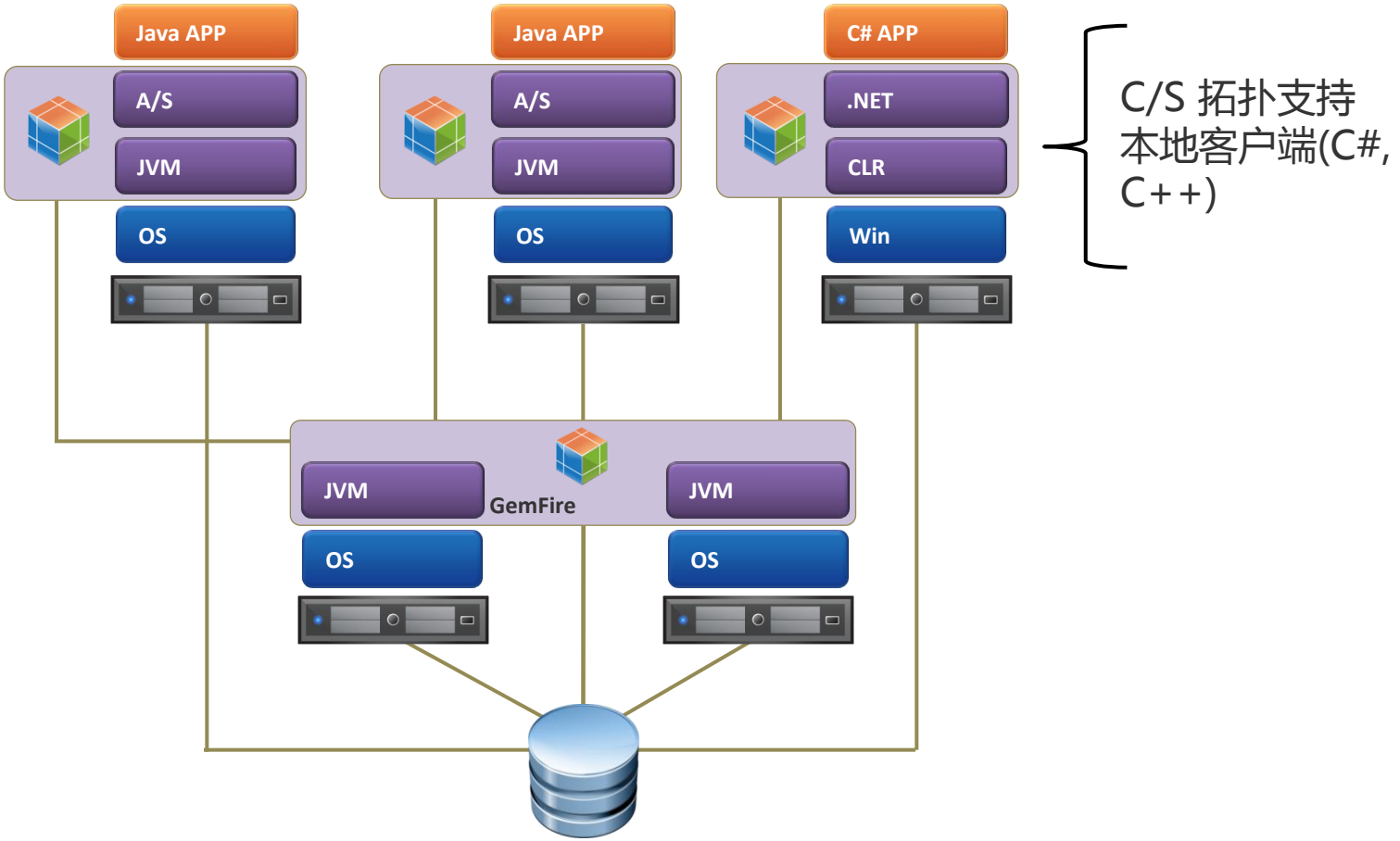
GemFire 主要拓扑结构

1. Peer-to-Peer
   - 缓存嵌入在应用程序中，与应用共享堆空间
   - Session 复制和L2 缓存



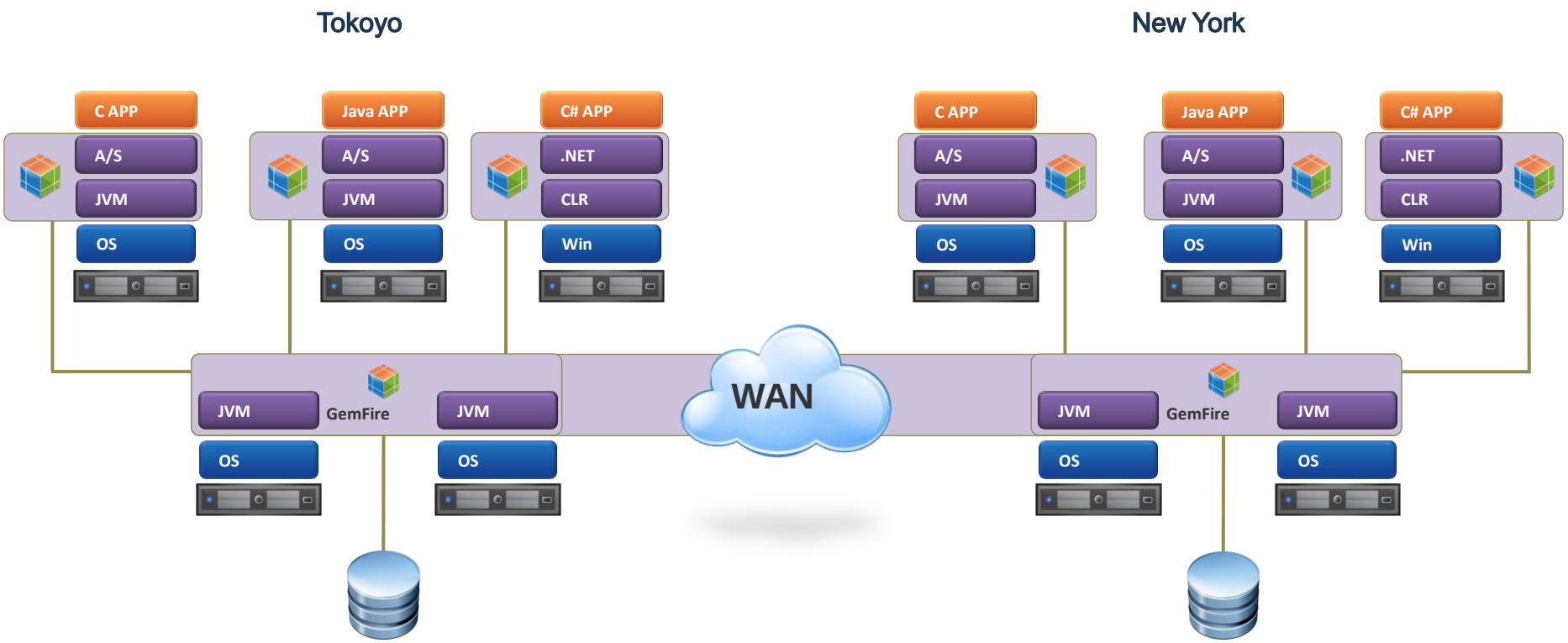
GemFire 拓扑结构

2. Client/Server
   - 中间缓存层由分布式集群系统来管理
   - Session 复制，L2 缓存， 数据关键应用， 容灾， 持续可靠性…

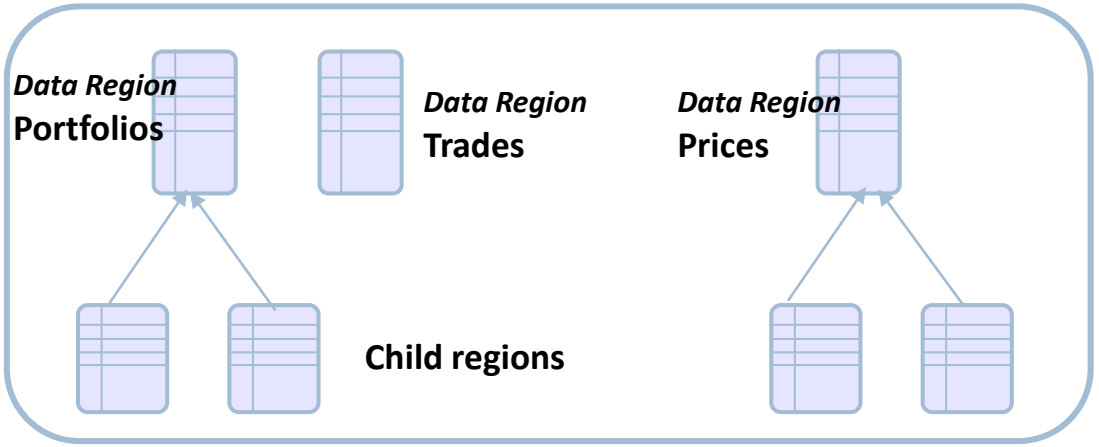


GemFire 拓扑结构

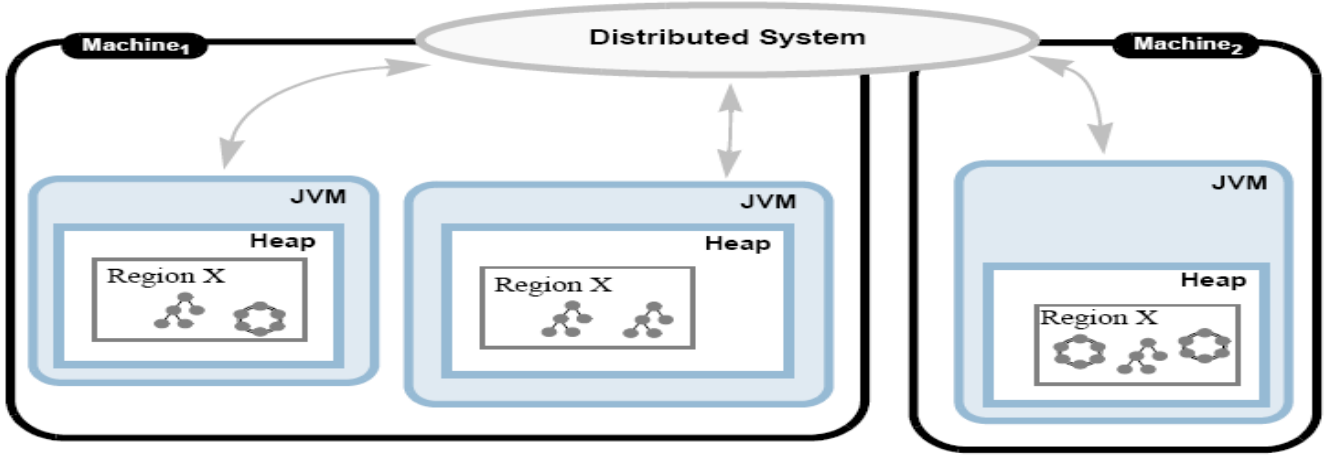
3. 多站点部署
   - 通过缓存数据在多站点之间同步，形成双向多活的数据中心。 如果任意一个站点的应用系统瘫痪，可以自动切换到其他站点继续运营



GemFire数据模型



- 单一的逻辑命名空间
- Regions是一个ConcurrentMap
  - 键值对结构



数据跨节点物理分布

Data Region主要特性

- Data Region 的主要特性
  - 溢出数据到磁盘(LRU)
  - 持久化到磁盘(逆向)
  - 跨集群节点进行数据同步
  - 分区 – 跨集群节点池化内存
  - 从外部数据源延迟加载数据
  - OQL语句

- Data Region 的主要操作
  - 同步读, 同步写, 异步写
  - 数据分布模型
    - D-no-ACK, D-ACK 和GLOBAL (有锁)

GemFire重要特性介绍

多站点部署	Gemfire集群可以跨地域部署在多个数据中心
数据分区	哈希分区,联位绑定
数据复制	数据多副本冗余
数据溢出	数据从内存写到临时文件上
数据持久化	数据从内存写到永久文件上
跨集群同步数据	通过Gateway Hub 跨WAN网进行数据同步
延迟加载数据	CacheLoader、JDBC
L2缓存数据	将数据缓存在内存中，使用LRU算法进行
异步读写数据	
数据重平衡	分布式系统通过此操作使各个节点的数据负载平均
事件流处理	MQ、CEP操作
分布式事务	支持分布式事务
分布式锁	支持分布式锁
类SQL语句	支持CQL查询
MapReduce	利用 Function 函数执行 Scatter-Gether并行处理
数据压缩和快照	将Region中的数据压缩后，导出到快照文件中备份

GemFire分布式事务-ACID

原子性和一致性

- MVCC事务控制
- Transaction State Thread
- Reservation System
- Failure Detection System

GemFire分布式事务-ACID

隔离

Process-based Thread:控制事务更新，提交后可见
  Volatile Read：无锁读，快速响应读操作
Transaction Write：基于事务写，如果事务用一个键写一个值，此时这个键已经被读了，后续的读将要返回事务引用。

持久性

Persistence：通过将内存中的数据写入永久性磁盘来保障。
Redundancy Copy：用于保障内存、网络和VM之间的可靠性。

GemFire分布式事务

对于分布式Region来说，数据经常跨多个缓存节点分布，但是事务只在一个缓存节点管理，因此可能多个region会参与到一个事务中。

事务操作首先作用在数据主拷贝节点，然后分布到其他成员，而不考虑是哪个成员先开始的缓存操作。

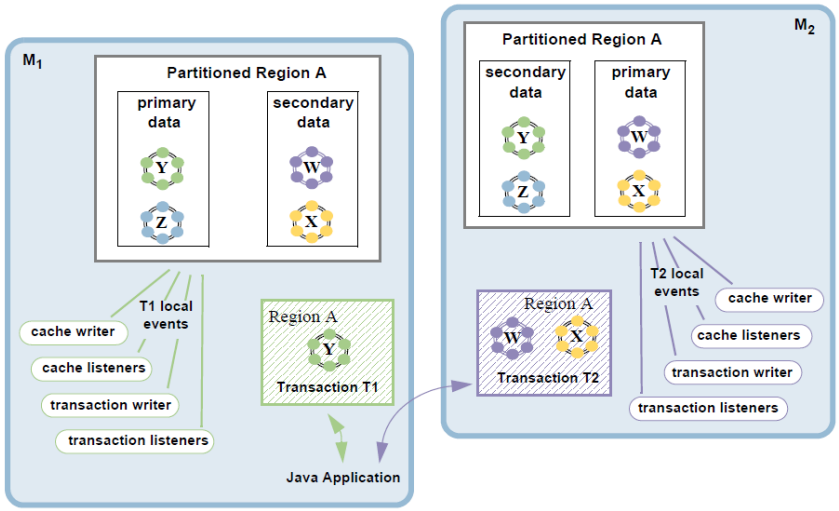
- 运行事务代码的成员被称为事务初始化器.
- 管理事务和数据的成员被称为事务数据节点

事务数据节点可能是事务初始化器也可能不是。当事务提交时，事务数据节点发布这份数据到其他节点。

GemFire分布式事务

分区区域的事务

M1运行着两个事务，T1和T2。T1作用在M1的Y数据上，因此M1既是事务初始化器也是事务数据节点；T2作用在M2的X和W数据上，因此M1是事务初始化器，M2是事务数据节点。



事务在数据节点上管理，包括事务视图、所有操作、所有本地缓存事件处理，当T2被提交时，M2的缓存被更新，事务事件跨系统来分布，好像此事务源于M2。

GemFire分布式事务

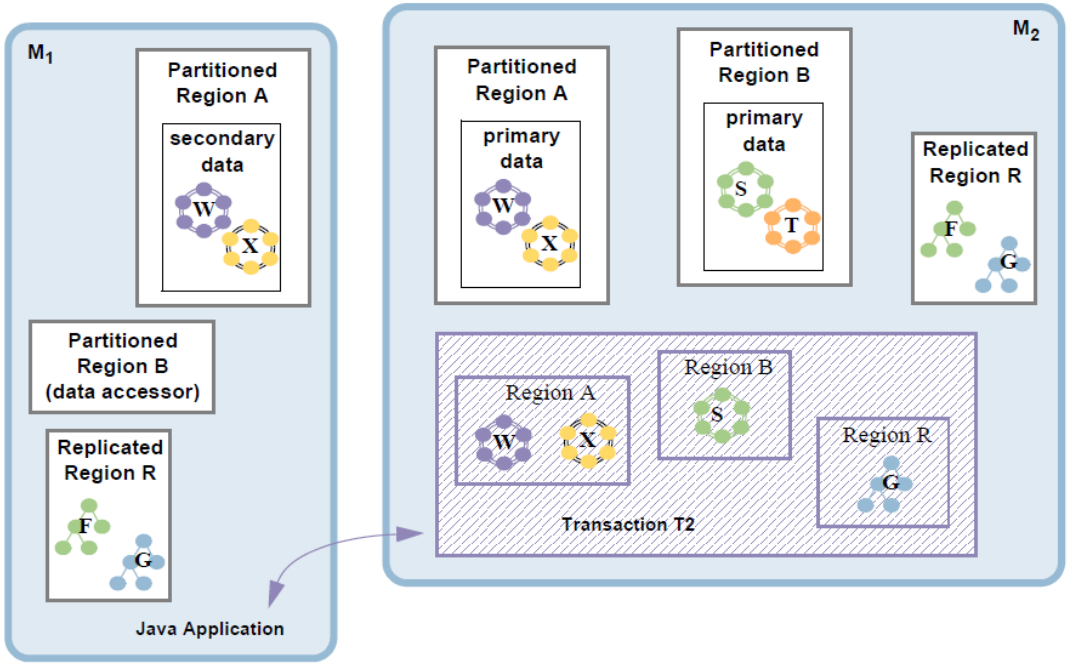
复制区域的事务

对于复制区域，事务和其操作被应用到本地成员，然后事务的状态被发布到其他成员，而数据如何传播，则根据region的属性设置。

① local—非分布式，在本地处理事务冲突. 与partitioned region不兼容

② distributed-no-ack—本地处理事务冲突，成员之间缺乏协调.

③ distributed-ack—即在本地处理事务冲突，又在成员之间处理事务冲突



分布式事务工作原理

Transaction view

在并发访问缓存的时候，事务之间是隔离的。每一个事务都有自己的私有视图，包括已读取的条目和其变更。当一个数据条目进入事务的时候，将在事务视图上生成一个数据状态的快照，此事务能够保存数据的原有状态，快照的另外一个用处是在提交阶段恢复写冲突。此事务维护着当前条目的视图，用来在事务中做出的变更。

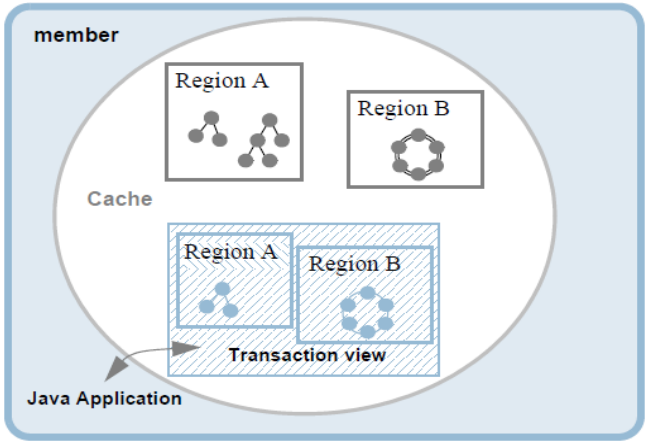
当提交成功时，在事务视图中记录的变更被合并到缓存上。如果提交失败或者事务回滚，所有的变更都将丢弃。
当事务提交时，事务管理系统使用“two-phase commit protocol”：

1. 通过驻留线程， 预留事务中所有的变更条目（对于分区区域，在事务运行的数据节点上做预留）
2. 在受影响的键上检查缓存，确保所有的数据条目都处于相同的状态。
- 如果有任何冲突，管理器回滚事务
- 如果没有冲突，管理器则执行如下步骤：

调用TransactionWriter，将事务更新写到本地缓存，传播更新到其他成员。在更新的缓存上，调用CacheListener，在事务运行的成员上调用TransactionListener。

3. 在条目上释放事务预留

管理器更新本地缓存，以非原子方式发布更新到其他成员。如果其他线程读取事务正在修改的键，他们可能是在预处理或者后处理的状态来查看。
如果是其他线程要修改事务正在修改的键，这个变更可能和事务更新混合在一起。当事务提交之后，此时事务处理的结果可能和原有的预期不一致。



GemFire分布式锁

GemFire分布式锁分为隐式锁和显式锁两种：

隐式锁：在分布式数据库系统中，大多数情况下自动地利用隐式锁来执行更新数据操作。

显式锁：在分布式数据库系统中，通过显式锁来锁定对数据库的操作，在涉及到多步分布式操作中，可以保证其原子性。

锁服务从系统成员处接受锁请求，并将其队列化，并按接受的顺序授予锁。锁授予者负责运行锁服务。锁授予者通常是系统成员之一。

GemFire锁授予者指定方式：

请求锁授予者状态的任何成员将被指定成锁授予者。这样在任意时间点上，做这个请求最近的成员将是锁授予者。

如果没有成员请求锁授予者状态，或者当前的锁授予者离开了，那么系统将从成员中指定一个为锁的授予者。

GemFire分布式锁

分布式锁创建

通过分布式锁服务来创建整个系统的分布式锁，在任何一个时间点上只允许一个线程拥有锁。另外线程将锁定整个服务，防止系统中其他线程锁定这个服务。

当分布式锁服务创建时，分布式系统中的某个成员将通过选举成为这个分布式锁服务的授予者。锁授予者负责管理这个锁。当这个成员出现故障时，锁授予功能将被迁移到其他的成员上，同时不丢失锁状态。

分布式锁释放

任何被离开成员所持有的锁会自动释放，对于剩下的成员仍然是可用的。

虽然锁授予者负责管理锁和队列化请求，如果锁授予者丢失，不会危及当前应用中已经授予的锁。其他的成员将被选举成为新的锁授予者，并迅速初始化获取当前系统中分布式锁的状态，之前等待锁的线程将被再次队列化和排序。

锁授予者根据接受的请求按顺序授予锁，以FCFG(先到先得)的方式授予锁进程。因为锁授予者不需要通过请求就能够获得锁，所以通常情况下我们会将需要最多锁的成员指定成锁授予者，这个成员可能有大量的隐式锁，也可能是有很多显式锁。

GemFire OQL查询语句

OQL类似与SQL，是OQL(ODMG)标准的一个子集，并基于SQL-92。但是用于查询复杂对象、对象属性和方法的查询语言。完全支持ASCII字符集和Unicode字符集。查询包含正常的查询和部分CQ查询。

- IMPORT、SELECT、FROM、WHERE、JOIN、IN、LIKE语句
- Region对象表达式
- 条件查询
- 属性可见

使用OQL查询，你能关联区域中成员的数据，并能够嵌套查询域对象。你也能使用索引来优化你的查询性能。

Select语句

基本的select语句格式为：

SELECT DISTINCT projectionList FROM collection1, [collection2, ...] [WHERE clause]

在Select语句中， FROM 语句数据放入Where语句中，Where语句在传递到select列表中过滤此数据。Select语句输出是列表操作的输出。类似于SQL对数据库的访问，from语句指定了数据表，where语句指定了行，select语句指定列。

From语句

From语句建立了对象的集合，在这些集合中对象的属性添加到命名空间范围。每一个FROM语句表达式必须赋值到一个集合. 这些表达式 /portfolios.keySet 是有效的，因为它赋值到一个集合中, 但是 /portfolios.名称赋值一个字符串将导致异常产生。在一个查询中，一个集合不仅包含了java.util.Collection的实现，也包含了

region path (evaluates to a Collection of entry values)
java.Util.Map (evaluates to a Collection of entries)
rray (Object or primitive arrays)

*如SQL查询, 在From语句中迭代整个表, 在From语句中，OQL语句遍历整个集合。*在如下的查询中, positions.values 赋值到集合中，因为positions是一个Map,同时Map上的方法值返回一个Collection.

```
IMPORT cacheRunner.Position;
SELECT DISTINCT "type"
FROM /portfolios, positions.values posnVal TYPE Position
WHERE posnVal.qty > 1000.00
```

如果positions 是一个List 而不是一个Map, 此查询将用于检索数据:

```
IMPORT cacheRunner.Position;
SELECT DISTINCT "type"
FROM /portfolios, positions posnVal TYPE Position
WHERE posnVal.qty >= 1000.00
```

Where语句

Where语句定义了搜索标准，where语句浓缩搜索用于过滤被from语句指定的集合。如果没有where语句，select将会接受整个collection或者collection集合。

查询处理器搜索匹配where条件的元素集合。如果在表达式上有一个索引通过where语句匹配，查询处理器可能使用检索来优化查询，因此避免了迭代整个集合。

WHERE 语句表达式为boolean条件， 为集合中的每个元素赋值。如果表达式为元素赋值为true, 查询处理器传递这个元素到select中. 本例子使用了WHERE语句返回带有类型xyz的portfolio对象。

```
SELECT DISTINCT * FROM /portfolios WHERE "type" = 'xyz‘
```

下面语句返回了所有带有xyz和active状态的portfolios 的集合。

```
SELECT DISTINCT * FROM /portfolios WHERE "type" = 'xyz' AND status = 'active'
```

关键字

Join关键字

如果在From语句的collection不互相关联，则where语句关联它们。
下面的语句将返回/Persons region中的所有的Persons和/Flowers region中的Flower。
SELECT DISTINCT p FROM /Persons p, /Flowers f WHERE p.name = f.name

索引支持region关联. 为了创建索引, 你将在两边的join条件上都创建索引。在语句查询的过程中，join关联被使用.
分区regions不支持region关联.

IN关键字

IN表达式为一个布尔值，提示了是否一个表达式存在于兼容类型的表达式集合内，如果e1和e2是表达式, e2 是一个collection, 同时e1是一个对象或者与e2是一个类型, 然后 e1 IN e2 是类型布尔的表达式. 则此表达式返回：

TRUE, 如果e1不是UNDEFINED 同时包含在collection e2中
FALSE, 如果 e1 不是 UNDEFINED 同时不包含在collection e2中
UNDEFINED, 如果 e1 is UNDEFINED

关键字

Like关键字
GemFire为Like关键字提供限制性支持,例如

where x like "<STRING>”

此语句where x like ‘abc‘与下列语句相同where x = 'abc‘

字符串或者是 ‘%’ 结尾，或者是 ‘*’ 结尾, 都表示为掩码. 这样此语句
where x like ‘abc%’与下列语句相同where x >= 'abc' and x < 'abd'

索引

索引为查询执行起到重要的作用。如果没有索引，查询将遍历集合中的每一个对象。如果设置索引之后，查询只遍历带有索引的集合，查询处理时间大大缩短。当索引数据变化时，索引也必须随之更新。

索引类型分为两种：

函数索引

GemFire提供了函数式排序索引，允许属性与常量之间进行比较，能够使用任意的关系型操作符来执行，并不限制属性和属性路径。这种比较能够引用引用参数、属性、属性路径或者将函数赋值给一个对象，来支持Comparable接口。

主键索引

GemFire支持主键索引，主键索引使用region自身的键。通过创建主键索引，查询服务必须关注Region键和值之间的关系，同时启用优化查询执行的关系。

查询服务并不自动关注region键和值之间的关系，因此你必须创建主键索引。

From语句必须是一个路径名，索引的表达式是一个生成键的表达式。例如，如果一个region有一个Portfolios作为值，而键是Portfolios的ID域，它们的索引表达式将为“id”。

主键索引不能排序。没有排序，为了在主键上获得一个排序的索引，在此属性上创建一个函数索引作为主键。

索引操作

创建一个索引

索引能够通过API或者通过声明cache.xml文件来创建。

索引创建需要在region上建立一个写锁，这样当索引创建的时候，put操作将被阻塞。

cache.xml法:

```
<region name="portfolios">
<region-attributes . . . >
<value-constraint>cacheRunner.Portfolio</value-constraint>
</region-attributes>
<index name="myFuncIndex">
<functional from-clause="/portfolios" expression="status"/>
</index>
<index name="myPrimIndex">
<primary-key field="id"/>
</index>
<entry>
```

API法：

```
qryService.createIndex("myFuncIndex", IndexType.FUNCTIONAL, "status", "/portfolios");
qryService.createIndex("myPrimIndex", IndexType.PRIMARY_KEY, "id", "/portfolios");
```

基本的Region访问

在语句中，Region的访问通过Region名称来指定，并在名称前加(/)。
/portfolios.name 返回 “portfolios”
/portfolios.name.length 返回10

也可以通过Region路径来访问条目的键和值
/portfolios.keySet 返回Region键的集合
/portfolios.entrySet 返回 Region条目的集合
/portfolios.values 返回Region条目的值集合
/portfolios 返回Region条目的值集合

Region对象表达式

当一个Region路径到一个Region中取值时，所引用的对象类型将有如下的类：

```
class QRegion
extends com.gemstone.gemfire.cache.Region, java.util.Collection {
}
```

Qregion包含了所有的Region对象的引用，同时实现了Region接口的所有方法。

```
SELECT DISTINCT positions FROM /portfolios
```

当你引用positions时，在这个查询中，/portfolios路径用于查询positions的集合

```
SELECT DISTINCT * FROM /portfolios.keySet
```

此语句将用于查询positions的所有键的集合

条件查询

OQL语句通过where关键字来指定查询的范围。

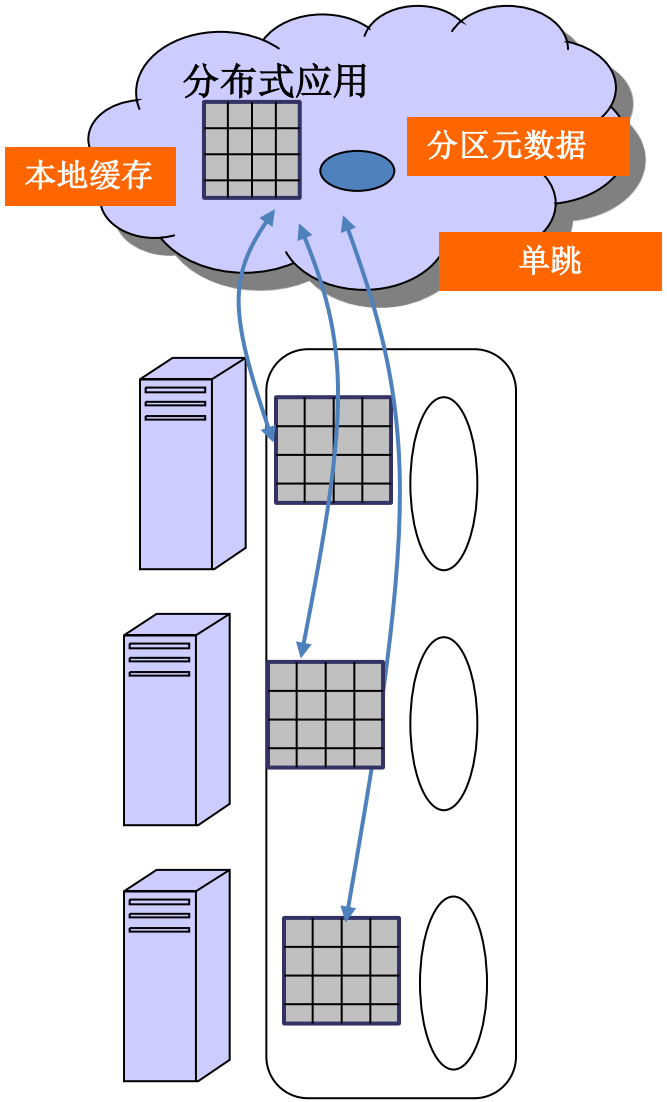
```
SELECT * FROM /portfolios WHERE status = 'active‘
```

FROM语句表达式在/portfolios中求取条目值的集合，这些值的属性被添加到语句的初始化范围，status在新范围中被解析。

```
IMPORT cacheRunner.Position;
SELECT DISTINCT "type"
FROM /portfolios, positions.values posnVal TYPE Position
WHERE posnVal.qty > 1000.00
```

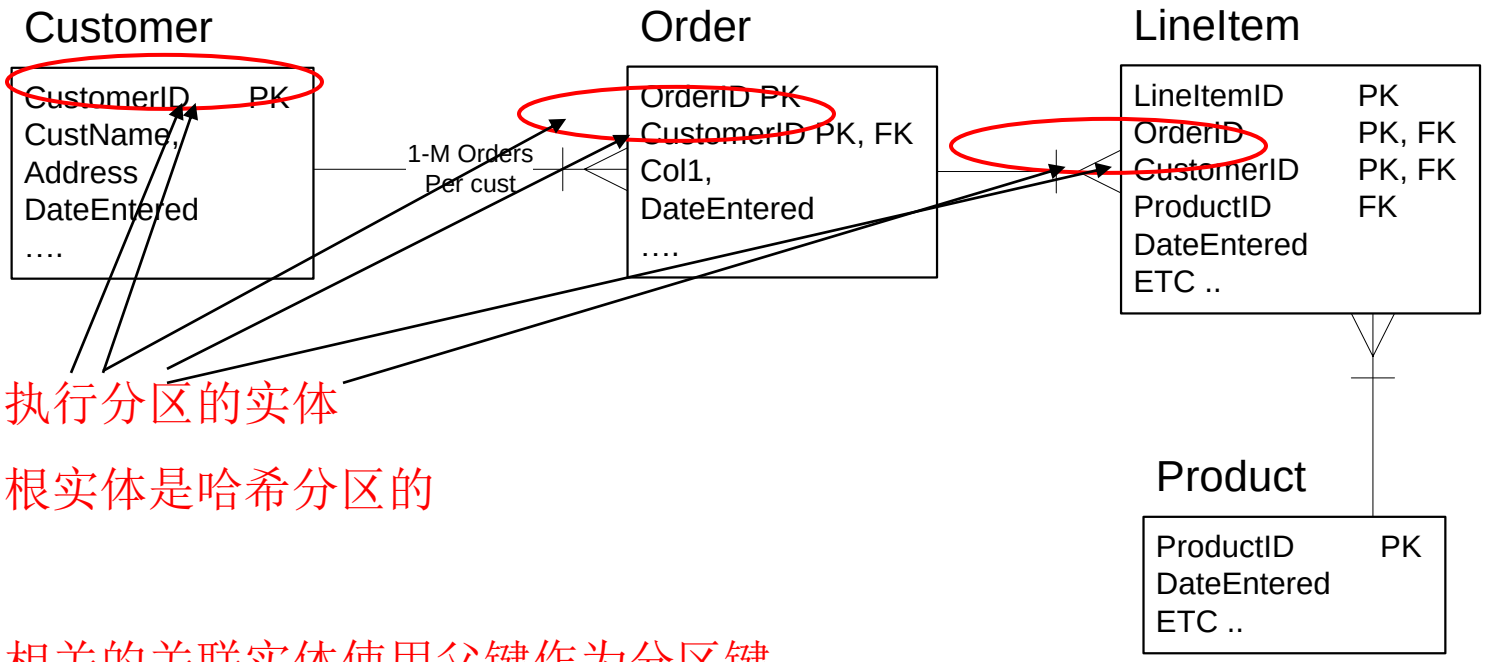
上面语句使用DISTINCT 作为关键字，用来过滤FROM语句表达式的集合，而WHERE条件中 更是限制posnVal.qty 的数量必须大于1000.00。

GemFire 数据分区策略-哈希分区



- 跨集群节点的水平分区
  - 将键值对均衡地分布到节点服务器上
- 一致性哈希算法
  - ***Hash(key)*** *– 映射到→* ***Bucket*** *– 映射到 →* ***Node***
- **GemFire**集群保证***Bucket-2-Node***源数据在所有节点上是一致的
- 当节点失败***Bucket*** *将被重新分配到新节点上*
- **Pros**
  - *get/put操作通常为一个网络跳*
  - 数据跨整个分布式集群做线性扩展

GemFire 数据分区策略-联位数据



执行分区的实体

根实体是哈希分区的

相关的关联实体使用父键作为分区键

水平分区数据 (并置有关联的数据)

- 适用于延时敏感的关键应用

GemFire 事件驱动框架

GemFire 通过对缓存应用的操作和异常状态生成事件，事件反映了成员、缓存、区域或数据条目的变化。你能够实现事件处理器来接受带有回调方法的事件，此回调事件指定某动作来响应每个事件。

- 事件循环

事件循环包括如下的步骤：

1. 开始一个操作，例如关闭缓存
2. 执行此操作生成如下对象：
   - 一个Operation类型对象，描述了触发此事件的方法
   - 一个事件对象，描述了此事件，例如成员和区域的来源
3. 转发事件给事件处理器
4. 当处理器接受事件时，将触发此处理器的回调方法
5. 如果操作是分布式的，则将转向其他成员进行操作，这些操作生成了自己的事件，通过监听器以相同的方式来处理

GemFire 事件对象

事件对象有若干个类型，依赖于具体操作。一些操作生成不同类型的多个对象。所有事件对象包含了描述事件的数据，每一个事件类型携带着不同的事件类型，而一个事件对象是固定的。

缓存事件和管理事件

Gemfire提供了两个事件和事件处理器的分类：

- 缓存事件- 提供了详细的通告，它们是应用处理事件的主要方法。
- 管理事件- 管理应用所使用的管理事件

两个事件类型能够在相同的成员上生成。一个操作能够生产管理事件、缓存事件、或者两者都有，依赖于此方法的实现。

应用能够使用事件处理器来触发回调。但是Gemfire单独维护缓存事件和管理事件的秩序，但是两种类型的事件不能同时使用。

CacheEvent

Caching API提供了6个事件类型：

RegionEvent 和 EntryEvent，扩展了CacheEvent 接口，是通用的事件，由 region 或者数据条目变化产生的事件。 RoleEvent扩展了RegionEvent。

另外，三个特定用途的事件，TransactionEvent由事务操作生成，，BridgeEvent有客户端和服务器之间操作生成和CqEvent由用户自定义的持续查询操作生成 ，这三个事件都代表了复杂事件。

所有的缓存事件都是同步的. 为了使事件异步，你可以利用另外的线程做回调操作。

CacheEvent

CacheEvent 接口定义了通用的事件方法. 事件对象包含了诊断事件情况的所需数据。他提供了有关事件的信息，包括缓存操作的描述，事件来源于哪个成员和区域，是否事件源于与事件处理器不同的缓存，同时回调参数传递到生成此事件的方法。此接口通过RegionEvent 和 EntryEvent 来扩展.

CacheListeners 和 CacheWriters 接受一个CacheEvent，或者是RegionEvent或者是EntryEvent. 相同的事件能够交付到更多的事件处理器，同时处理器能够对不同的动作做出响应。 CacheListeners在事件生

RegionEvent和EntryEvent

RegionEvent

RegionEvent接口扩展了CacheEvent.它提供了影响整个Region的操作信息。RegionEvents将传递到CacheListener,RegionMembershipListener,RegionRoleListener和CacheWriter。

EntryEvent

EntryEvent包含了影响数据条目的操作信息，包括key，value(before)，value(after).对于分区Region。如果获取数据条目的成本可能太高，则”Entry not available”可能发生，例如，此数据条目已经持久化到磁盘上。

你能够从EntryEvent中查询序列化的值。EntryEvent也提供了transaction ID.如果操作发生在事务之外，则ID为Null。

EntryEvents传递到CacheListener,RegionMembershipListener,RegionRoleListener和CacheWriter。另外，在TransactionEvent中，TransactionListener接受一个EntryEvent实例列表。

RoleEvent、TransactionEvent和BridgeMembershipEvent

RoleEvent

RoleEvent接口扩展了RegionEvent。事件包括成员关系角色加入或者离开的信息，因此影响着Region的可靠性. RoleEvents被传递到RegionRoleListener。

TransactionEvent

TransactionEvent描述了事务所做的工作。依赖于事件处理器，此工作即将被提交，正在提交或者通过回滚放弃或者提交失败。此工作通过EntryEvent 实例的列表来表示。列表中的条目事件列出了哪些操作在事务中被执行。当事务操作被执行，事件被合并，只有最后的事件仍然存在列表之中。因此，如果条目A被修改，然后条目B，然后在条目A，列表将包含条目B的事件，紧跟着条目A的事件。

TransactionEvent有获得 TransactionId的方法，同时返回CacheEvents的列表，按操作顺序排列。TransactionEvents被传递到TransactionWriter和TransactionListener。

BridgeMembershipEvent

BridgeMembershipEvent提供客户端或服务器变化的信息。此事件方法允许你为其他进程获得唯一ID，确定此事件是否影响客户端的连接。BridgeMembershipEvents传递给BridgeMembershipListener。

CqEvent和GatewayEvent

CqEvent

CqEvent提供了持续查询的事件信息。CqEvent通过客户端创建，运行在服务器上用于过滤服务器事件。服务器运行与此事件关联的查询，然后转发这个事件。

CqEvents被传递给CqListener。

GatewayEvent

GatewayEvent被传递给网关队列，用于批处理，异步传递给GatewayEventListener。在区域中这些是都是条目事件的子集。

GemFire MapReduce批处理

不足之处: 计算任务花费大量时间来获取数据

扩展数据操作

1) 路由数据操作到数据所在地

2) 在每个分区上做并行处理

将操作map到多个分区中

在每个分区中运行并行应用处理

reduce处理结果

```
@Override
public void execute(FunctionContext fc) {

    RegionFunctionContext context = (RegionFunctionContext)fc;
    String sWhere = context.getArguments().toString();
    SelectResults resultSet = null;
    try {
        Region region = PartitionRegionHelper.getLocalDataForContext(context);
        resultSet = region.query("");
    } catch (FunctionDomainException e2) {
        e2.printStackTrace();
    } catch (TypeMismatchException e2) {
        e2.printStackTrace();
    } catch (NameResolutionException e2) {
        e2.printStackTrace();
    } catch (QueryInvocationTargetException e2) {
        e2.printStackTrace();
    }

    context.getResultSender().lastResult((Serializable)resultSet);
```

12306成长历程

12306网站3年时间访问量从10亿PV暴涨到100亿PV，售票量从100万增加到500万，出票处理能力200张/秒增加到1000张/秒。

	尖峰日 PV值	放票次数	网络带宽	尖峰日售票	同时在线人数限制	处理订单(张/秒)
2012	10亿	4次	1.5G	110万	1万	200
2013	15亿	10次	3G	265万	20万	450
2014	144亿	16次	5G	501万		1000
2015	297亿	21次	12G	564万		1032

12306的双高挑战

12306网站曾被认为是“全球最忙碌的网站”，在应对高并发访问处理方面，曾备受网民诟病。在2015年春运高峰日的PV值是297亿，流量增加1000倍。

12306作为一个面向公众的系统,互联网售票系统具备所有大型互联网系统的特性:”高并发”,”高流量”。尤其是在我国的各个节假日,系统的访问 量会激增,导致整个系统后台压力过大,响应变慢。

参考目前国内外成熟电商网站经验,例如国内的淘宝,百度,国外 的谷歌等这类有着海量用户访问的网站,都不约而同的采用分布式的数据处理平台,而Gemfire正是一个基于内存的分布式数据库,并且拥有大 量的成功案例,非常适合来解决这样的问题。

12306高并发挑战

1.海量的用户访问(2012年春运高峰为13亿/天,可推算出最高并发约为3万/ 秒)

- Gemfire本身是基于内存的技术架构,对于并发访问有天然的IO优势
- 同时Gemfire是一个分布式数据库,可以将数据访问的请求分散到集群中,有效降低了单个服务器的负荷
- 动态的分布式架构,可以在运行时增加服务节点,从而获得更高的并发性能

12306高流量挑战

2.低延迟的余票查询

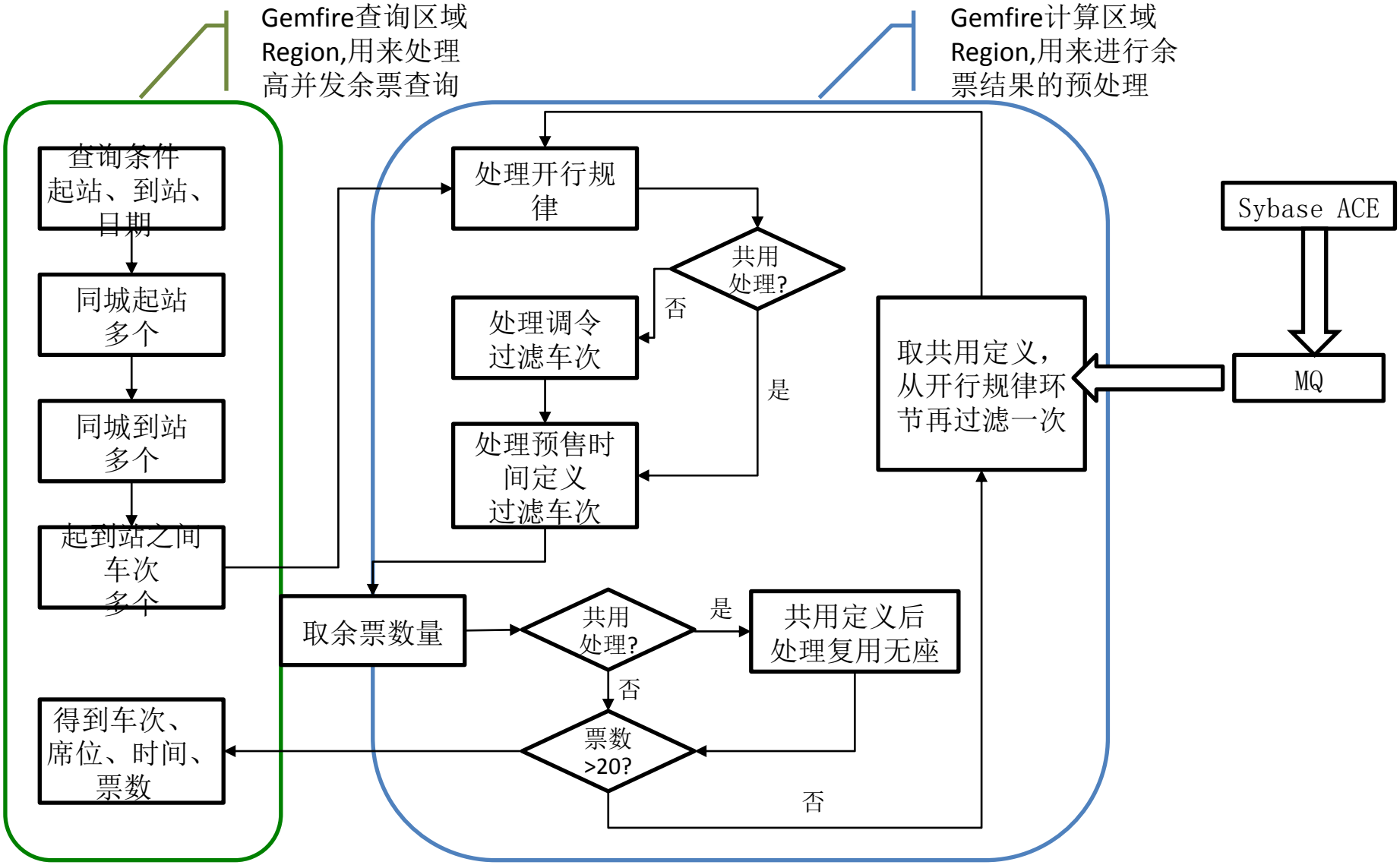
- Gemfire支持类Map-Reduce并行处理
- 按车次将余票、共用定义等数据拆分成多个独立的计算单元,对余票查询中最耗时的共 用定义部分做预先处理,生成查询缓存
- 余票数据变化时,动态更新查询缓存
- 有了预处理及数据同步过程维护的动态查询缓存,单次查询可以控制在10ms-300ms之间, 同时10分钟的固有延迟也不存在了

12306数据同步挑战

1.余票数据更新需要实时同步到Gemfire中

- Gemfire有设计良好的事件处理架构
- Sybase服务器提供了异构系统的消息服务
- Sybase数据库的数据更新以消息事件的形式发送给Gemfire

Gemfire 缓存数据逻辑架构

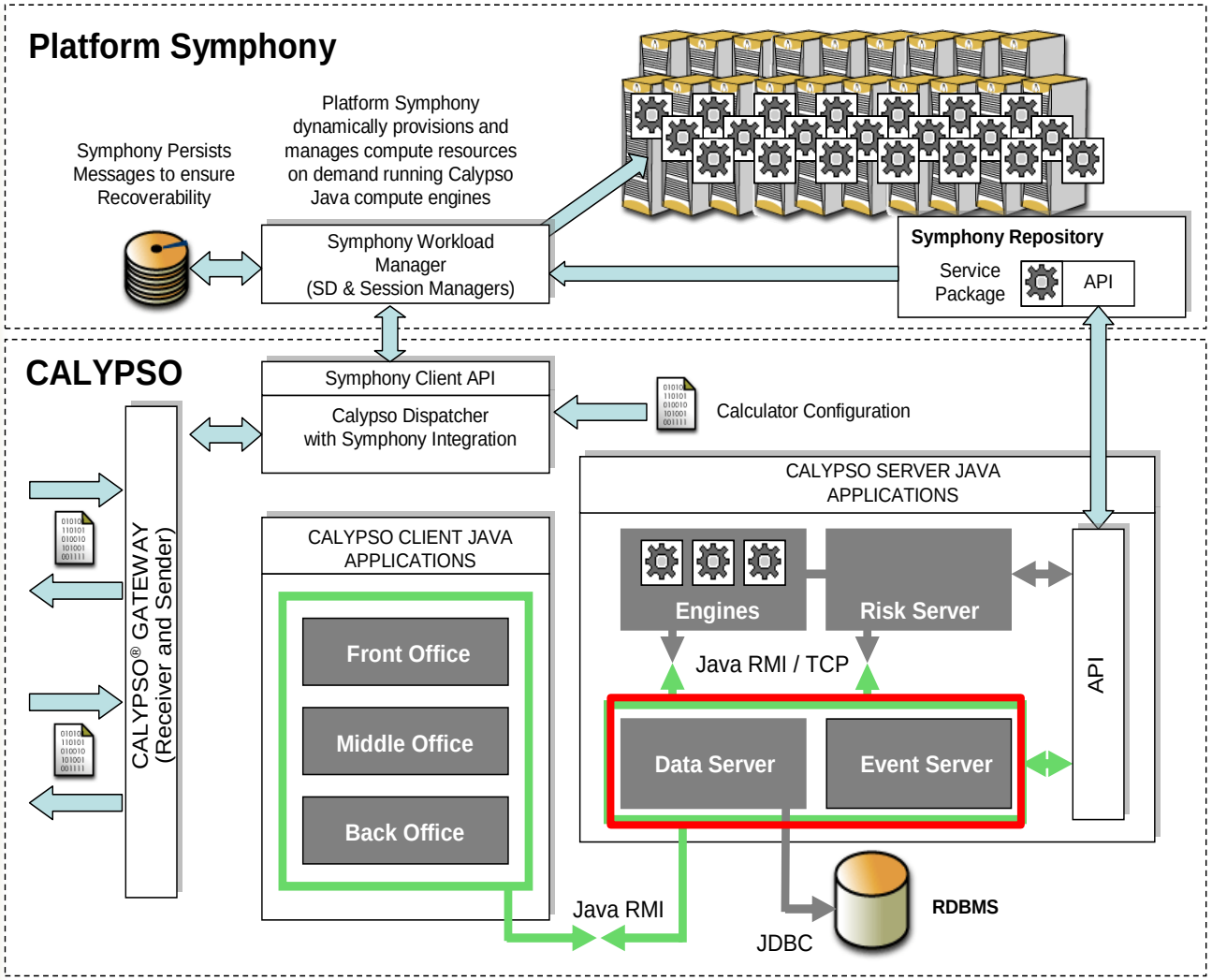


Gemfire 在金融领域的应用

某国际大型投行的衍生品交易系统整体架构：

平台核心组件包括：

- Calypso – 衍生品交易平台
- Platform Symphony – 任务调度
- BEA JRockit – 垃圾收集
- GemFire – 数据服务器和事件服务器



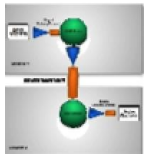
应用场景分享

- 存储
- 持久化
- 事务
- 查询

- 高可靠
- 负载均衡
- 数据复制
- L1 缓存

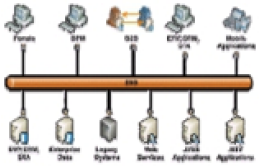
+ 消息系统

- 数据分布式
- 事件传播
- 高可靠投递



+ 服务总线

- 系统集成
- 数据传输
- 总耦合服务

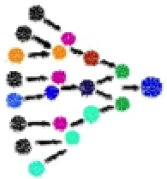


+ 网格控制器

- 任务分解
- 分布式任务指定
- Map-Reduce, Scatter-Gather
- 结果汇总

+ 复杂事件处理

- 业务事件检测
- 实时分析
- 事件驱动架构




ChinaUnix
ITPUB
THANKS