VMware vFabric SQLFire

- Main Memory
- Distributed (cloud, commodity)
  - Partitioning
  - (A)synchronous Replication
- Stored Procedures
- Closed Source

Familiar, SQL-like implementation of a distributed database system

Unless otherwise noted, info and examples are from VMware's documentation.
Operates In

VMware vFabric SQLFire
- Main Memory
- Distributed (cloud, commodity)
- Failovering
- Asynchronous Replication
- Tiled Procedures
- Closed Source

Familiar, SQL-like implementation of a distributed database system

SQL on Fire

VMware vFabric
- VMware’s suite of software for cloud application development
- SQLFire = GemFire (at right)
- Newer, Standard SQL, Optimized

Executes Across
VMware vFabric

- VMware's suite of software for cloud application development
- SQLFire ≈ GemFire (at right)
  - Newer, Standard SQL, Optimized
vFabric Licensing: per VM, average

<table>
<thead>
<tr>
<th>Cloud Application Requirement</th>
<th>vFabric Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deploy applications on pools of virtual infrastructure rather than physical servers</td>
<td>Per-VM pricing for hardware independence</td>
</tr>
<tr>
<td>Accommodate workload spikes from business cycles coupled with large user base</td>
<td>License based on average, not peak usage. Usage tracked but not limited, in order to accommodate workload spikes.</td>
</tr>
<tr>
<td>Reduce time-to-market by initially releasing applications in a “good enough” configuration that is later optimized as performance data is collected</td>
<td>Re-use licenses across different application tiers: web tier, app servers, data caches, databases, and message servers. For instance, you can initially deploy 10 vFabric Advanced VMs as application servers, then later re-deploy them as 10 database servers.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>vFabric Edition</th>
<th>Price with Production Support (USD)</th>
<th>Price with Basic Support (USD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>vFabric Standard</td>
<td>$1,200/VM + 25% annual SnS</td>
<td>$1,200/VM + 21% annual SnS</td>
</tr>
<tr>
<td>vFabric Advanced</td>
<td>$1,800/VM + 25% annual SnS</td>
<td>$1,800/VM + 21% annual SnS</td>
</tr>
</tbody>
</table>
VMWare

Focuses:
- Virtualization
- Distributed applications

Location: Palo Alto, CA
Sales: $3.767096 bil
Profits: $0.723936 bil
Assets: $8.680808 bil
Employees: 11,000

Contains
SQLFire members

- Data Stores (majority)
  - Host data
  - Execute local/distributed sqlf queries
  - Single-hop access to any piece of data
- Accessors
  - Do not host data
  - Execute local/distributed sqlf queries
  - Single-hop access to any piece of data
- Locators
  - Do not host data
  - Do not touch any queries
  - Discover members of cluster
  - Clients query the locator for the server with the least amount of load (other active client connections)
    - Only way to balance server load from clients
Partitions and Replicates Data
CREATE TABLE AIRLINES
(
    AIRLINE CHAR(2) NOT NULL CONSTRAINT AIRLINES_PK PRIMARY KEY,
    AIRLINE_FULL VARCHAR(24),
    ECONOMY_SEATS INTEGER,
    BUSINESS_SEATS INTEGER,
    FIRSTCLASS_SEATS INTEGER
)

CREATE TABLE FLIGHTS
(
    FLIGHT_ID CHAR(6) NOT NULL,
    ORIG_AIRPORT CHAR(3),
    DEPART_TIME TIME,
    DEST_AIRPORT CHAR(3),
    ARRIVE_TIME TIME,
    MILES INTEGER,
    AIRCRAFT VARCHAR(6),
    CONSTRAINT FLIGHTS_PK PRIMARY KEY (FLIGHT_ID)
)

Our developers are comfortable with SQL. Operating in the cloud, they now need to easily and efficiently:
- Partition large datasets
- Replicate data to increase throughput and guard against (isolated) node failures
- Remember, working in main memory

http://www.infoq.com/news/2012/01/sqlfire-1-0
easily and efficiently:
- Partition large datasets
- Replicate data to increase throughput and guard against (isolated) node failures
- Remember, working in main memory

```sql
CREATE TABLE FLIGHTS (
    FLIGHT_ID INT PRIMARY KEY,
    DEPARTURE_TIME TIME,
    DEPARTURE_MILES INTEGER,
    DEPARTURE_AIRCRAFT VARCHAR(6),
    ARRIVE_TIME TIME,
    MILES INTEGER,
    AIRCRAFT VARCHAR(6),
    CONSTRAINT FLIGHTS_PK PRIMARY KEY (FLIGHT_ID)
)
```

[Image]
CREATE TABLE AIRLINES
(
AIRLINE Char(2) NOT NULL CONSTRAINT AIRLINES_PK PRIMARY KEY,
AIRLINE_FULL VARCHAR(24),
ECONOMY_SEATS INTEGER,
BUSINESS_SEATS INTEGER,
FIRSTCLASS_SEATS INTEGER
) REPLICATE;

Replication handled synchronously (blocking)

CREATE TABLE FLIGHTS
(
FLIGHT_ID CHAR(6) NOT NULL,
ORIG_AIRPORT CHAR(3),
DEPART_TIME TIME,
DEST_AIRPORT CHAR(3),
ARRIVE_TIME TIME,
MILES INTEGER,
AIRCRAFT VARCHAR(6),
CONSTRAINT FLIGHTS_PK PRIMARY KEY (FLIGHT_ID)
)
PARTITION BY COLUMN (FLIGHT_ID);

CREATE TABLE FLIGHTAVAILABILITY
(
FLIGHT_ID CHAR(6) NOT NULL,
SEGMENT_NUMBER INTEGER NOT NULL,
FLIGHT_DATE DATE NOT NULL,
ECONOMY_SEATS_TAKEN INTEGER DEFAULT 0,
BUSINESS_SEATS_TAKEN INTEGER DEFAULT 0,
FIRSTCLASS_SEATS_TAKEN INTEGER DEFAULT 0,
CONSTRAINT FLIGHTAVAIL_PK PRIMARY KEY (FLIGHT_ID,
SEGMENT_NUMBER,
FLIGHT_DATE),
CONSTRAINT FLIGHTS_FK2 Foreign Key (FLIGHT_ID,
SEGMENT_NUMBER)
REFERENCES FLIGHTS (FLIGHT_ID,
SEGMENT_NUMBER)
)
PARTITION BY COLUMN (FLIGHT_ID)
COLOCATE WITH (FLIGHTS);

http://www.infoq.com/news/2012/01/sqlfire-1-0
and documentation
CREATE TABLE COUNTRIES
(
    COUNTRY VARCHAR(26) NOT NULL,
    COUNTRY_ISO_CODE CHAR(2) NOT PRIMARY KEY,
    REGION VARCHAR(26),
)
SERVER GROUPS (OrdersDB, OrdersReplicationGrp)

Partitioning/Replication applied within server group

Multiple server groups for logical partitioning

Logically partition your data into multiple schemas. Associate each schema with a server group. For instance, for a financial trading application, all trades, positions and pricing data could be managed in Group1, and all reference data can be managed in Group 2. You can add or remove capacity to any group as needed.
Executes Across Partitions
CallableStatement callableStmt = connection.prepareCall("{CALL order_credit_check(?) }");
callableStmt.setArray(1, <list of customer IDs>);

// SQLFire data-aware procedure invocation
CallableStatement callableStmt = connection.prepareCall("{CALL order_credit_check() 
+ "ON TABLE Orders WHERE customerID IN (?)}");
callableStmt.setArray(1, <list of customer IDs>);

// order_credit_check will be executed in parallel on all members where the orders
// corresponding to the customerIDs are managed
Partitions

Multiple server groups for logical partitioning

Logically partition your data into multiple schemas. Associate each schema with a server group.

For instance, for a financial trading application, all trades, positions, and pricing data could be managed in Group 1, and all reference data can be managed in Group 2.

You can add or remove capacity to any group as needed.

Executive of Stored Procedures

```java
connection.prepareStatement("CALL order_credit_check(? ?);", "customer IDs");
```

Preparation of invocation

```java
stmt = connection.prepareStatement("CALL order_credit_check() " + "WHERE customerID IN (?);", "id of customer IDs");
```

Execution in parallel on all members where the orders are managed

Alternatively,

```java
stmt.executeUpdate();
```

Preparation of execution
Alternatively Partitioned

Partitioning Schemes Supported

```
CREATE TABLE Orders
(
    OrderId INT NOT NULL,
    ItemId INT,
    NumItems INT,
    CustomerName VARCHAR(100),
    OrderDate DATE,
    Priority INT,
    Status CHAR(10),
    CONSTRAINT fk_Orders PRIMARY KEY (OrderId)
)
PARTITION BY COLUMN (CustomerName)
GROUPING (OrderDate)
```

```
CREATE TABLE Orders
(
    OrderId INT NOT NULL,
    ItemId INT,
    NumItems INT,
    CustomerName VARCHAR(100),
    OrderDate DATE,
    Priority INT,
    Status CHAR(10),
    CONSTRAINT fk_Orders PRIMARY KEY (OrderId)
)
PARTITION BY RANGE (Priority)
```

```
CREATE TABLE Orders
(
    OrderId INT NOT NULL,
    ItemId INT,
    NumItems INT,
    CustomerName VARCHAR(100),
    OrderDate DATE,
    Priority INT,
    Status CHAR(10),
    CONSTRAINT fk_Orders PRIMARY KEY (OrderId)
)
PARTITION BY VALUES (Priority)
```
Partitioning Schemes Supported

CREATE TABLE Orders
(
    OrderId INT NOT NULL,
    ItemId INT,
    NumItems INT,
    CustomerName VARCHAR(100),
    OrderDate DATE,
    Priority INT,
    Status CHAR(10),
CONSTRAINT Pk_Orders PRIMARY KEY (OrderId)
)
PARTITION BY COLUMN (CustomerName)
SERVER GROUPS (OrdersDBServers);

CREATE TABLE Orders
(
    OrderId INT NOT NULL,
    ItemId INT,
    NumItems INT,
    CustomerName VARCHAR(100),
    OrderDate DATE,
    Priority INT,
    Status CHAR(10),
CONSTRAINT Pk_Orders PRIMARY KEY (OrderId)
)
PARTITION BY LIST (Status)
VALUES ('pending', 'returned'),
VALUES ('shipped', 'received'),
VALUES ('hold');

CREATE TABLE Orders
(
    OrderId INT NOT NULL,
    ItemId INT,
    NumItems INT,
    CustomerName VARCHAR(100),
    OrderDate DATE,
    Priority INT,
    Status CHAR(10),
CONSTRAINT Pk_Orders PRIMARY KEY (OrderId)
)
PARTITION BY (MONTH(OrderDate));

CREATE TABLE Orders
(
    OrderId INT NOT NULL,
    ItemId INT,
    NumItems INT,
    CustomerName VARCHAR(100),
    OrderDate DATE,
    Priority INT,
    Status CHAR(10),
CONSTRAINT Pk_Orders PRIMARY KEY (OrderId)
)
PARTITION BY RANGE (Priority)
VALUES BETWEEN 1 AND 11,
VALUES BETWEEN 11 AND 31,
VALUES BETWEEN 31 AND 50;

CREATE TABLE COUNTRIES
(
    COUNTRY VARCHAR(26) NOT NULL,
    COUNTRY_ISO_CODE CHAR(2) NOT PRIMARY KEY,
    REGION VARCHAR(26),
)
REDUNDANCY 1
Any peer or server detects problem; issues suspect alert to membership manager. After timeout, manager propagates revised membership list.
Overflow

Persitence

Can use disk
CREATE TABLE table-name {
    ( { column-definition | table-constraint } 
    [, { column-definition | table-constraint } ] *) 

    [( column-name [, column-name ] * ) ]

    AS query-expression
    WITH NO DATA
}

[ partitioning_clause | REPPLICATE ]
[ SERVER GROUPS ( server_group_name [, server_group_name ]*)]
[ HUB ( 'hub-name' | ALL )]
[ ASYNCEVENTLISTENER (async-listener-id) ]
[ EVICTION BY {eviction_criterion} EVICTACTION { OVERFLOW | DESTROY }]
[ EXPIRE { TABLE | ENTRY } WITH { IDLETIME value | TIMETOLIVE value}]
ACTION { DESTROY | INVALIDATE }]*

[ PERSISTENT ] [ 'disk-store-name' ] [ ASYNCHRONOUS | SYNCHRONOUS ]
Overflow

Can use disk

Persistence
CREATE TABLE COUNTRIES
(
  COUNTRY VARCHAR(26) NOT NULL CONSTRAINT COUNTRIES_UNQ_NM Unique,
  COUNTRY_ISO_CODE CHAR(2) NOT NULL
CONSTRAINT COUNTRIES_PK PRIMARY KEY,
  REGION VARCHAR(26),
CONSTRAINT COUNTRIES_UC CHECK (country_ISO_code = upper(country_ISO_code)) )
) REPLICATE PERSISTENT;
-- uses default diskstore

CREATE DISKSTORE STORE1 MAXLOGSIZE 1024
AUTOCOMPACT TRUE
ALLOWFORCECOMPACT TRUE
COMPACTIONTHRESHOLD 80
TIMEINTERVAL 223344
WRITEBUFFERSIZE 19292393
QUEUESIZE 17374 'dir1'(456)

CREATE TABLE Orders(OrderId INT NOT NULL,ItemId INT )
persistent 'OrdersDiskStore' asynchronous
VMware vFabric SQLFire

- Main Memory
- Distributed (cloud, commodity)
  - Partitioning
  - (A)synchronous Replication
- Stored Procedures
- Closed Source

Familiar, SQL-like implementation of a distributed database system

Unless otherwise noted, info and examples are from VMware’s documentation.