Comment faire tourner une JVM de 16 To

Speaker : Antoine Chambille et Romain Colle (Quartet FS)
Format : Conférence
Date : 22 avril 2016

Il y'a un an, un client a lancé un défit à Quartet : faire tourner une JVM avec 16 Go.
Quartet FS développe ActivePivot, une solution Java d'analyse de données.

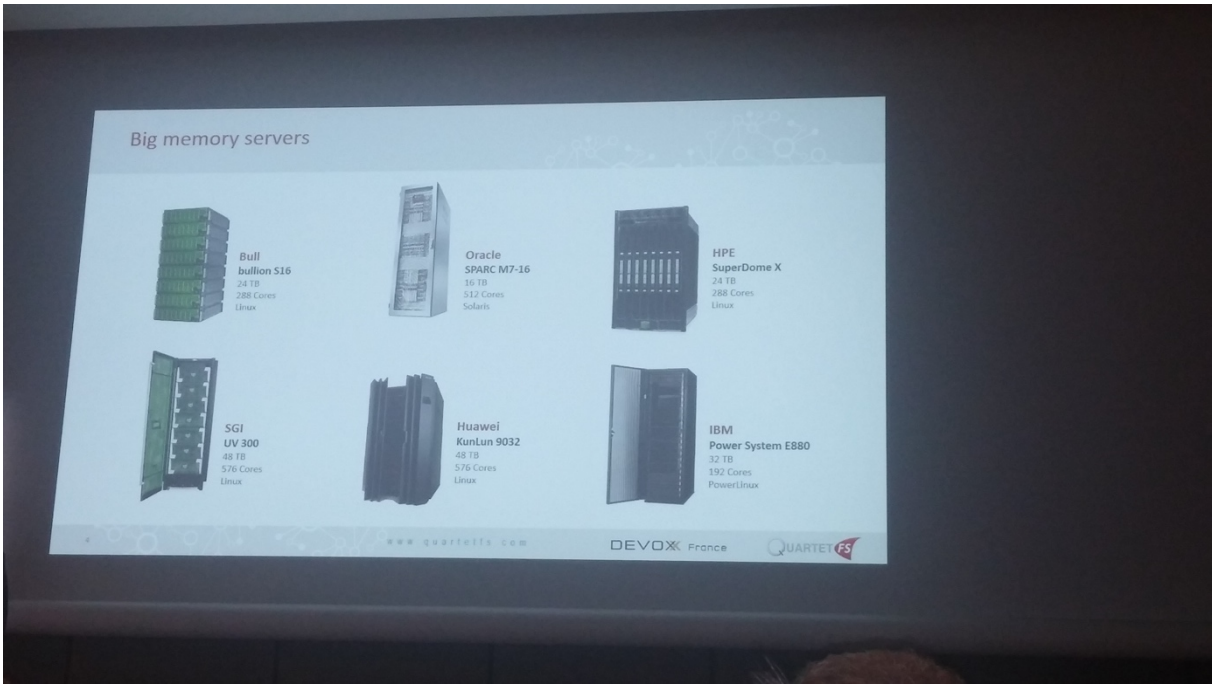
Contexte métier

Application financière calculant le risque de crédits. Repose sur des simulations de condition de marchés. 100 aine de milliards de valeurs. 8 TB de data. Ancienne approche : traitements batchs qui tournent la nuit. Solution remise en cause par la réglementation. Les banques veulent aller plus loin que des analyses quotidiennes.
Problématique : agrégation de données sur de grands volumes de données.
ActivePivot est une solution :

- In-Memory Database
  - Column Store
  - Designed for multi-core
  - Multimentional Aggregation
  - High Cardinality bitmap index
- Développée en Java

Big memory servers

- Bullion 24 TB – 288 Cores – Linux : serveur modulaire qu'on peut étendre, assemblé en France
- Oracle Sparc M7-16 : 16 TB – 512 Cores – Solaris
- Huawai 48 TB, 576 Cores – Linux



Avec l'arrivée des barrettes mémoire de 128 Go, leur capacité devrait doubler d'ici l'année dernière.

ActivePivot : 1, 2, 4 To de données en temps réel.

Possibilité de sortir des données en offheap :

1. Gestion off-heap : possible depuis Java 1.4 avec NIO (java.nio.Buffer)
DoubleBuffer buf = ByteBuffer.allLocateDirect
   - ⇨ Allocation mémoire

2. Utilisation de la classe sun.misc.UNSAGE
   - ⇨ Très bien pour des données primitives, pas pour des objets sérialisés : column of numbers, index, hash tables, vectors of simulations

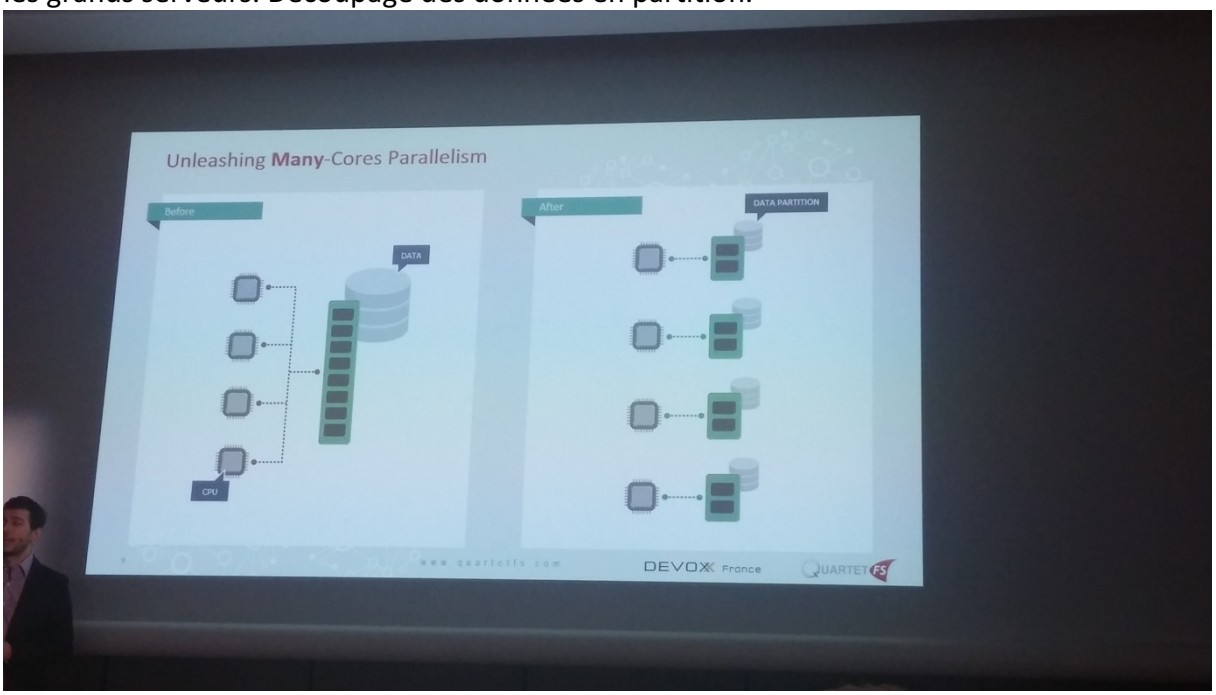
Sur 16 To : 3 TB de Heap for queries and calculations et 12 TB Off-Heap
Nécessite du calcul en parallèle

Ils adressent depuis longtemps le problème du « many pool »
- ⇨ Utilisation avant Java 7 du Fork/Join Pool
- ⇨ Techniques de programmation lock free pour les structures :
  - Dictionnaires, indexes, queues réécrites
- ⇨ Les locks ont été retirés lors des transactions bases de données

Ces techniques ne suffisent plus à partir de 32 cœurs. Aujourd'hui, les grands serveurs disposent de 100aine de cœurs.

Il y'a 3 ans, ActivePivot est reparti de zéro.

L’architecture ActivePivot a été revue pour exploiter le fait que 1 cœur a sa mémoire dans les grands serveurs. Découpage des données en partition.



Il font attention où ils mettent leurs données.
Architecture NUMA : lorsqu’un processeur accède à la mémoire d’un autre cœur, les perfs se dégradent. La bande passante mémoire est déterminante pour ActivePivot.
Si NUMA pas prise en compte, performances à auteur d’un laptop.

Comment supporter NUMA en Java ?

Lorsqu’un process alloue de la mémoire, c’est l’OS qui décide où allouer la mémoire.
Sur Linux et Solaris, la mémoire est allouée à côté du processeur.
Pré-requis : détecter la topologie NUMA du système.
Les OS fournissent des API. Utilisation de JNA pour accéder à la librairie (ex : libnuma et pthread sur Linux). Muni de l’identifiant du thread pour récupérer le nœud numa. Et attacher le thread à ce nœud.

Comment répartir les nœuds ?
Choisir le thead-pool en fonction des données qu’exploitent la requête.

3000 threads. Sur SPARC, utilisation de 8 threads par cœur.

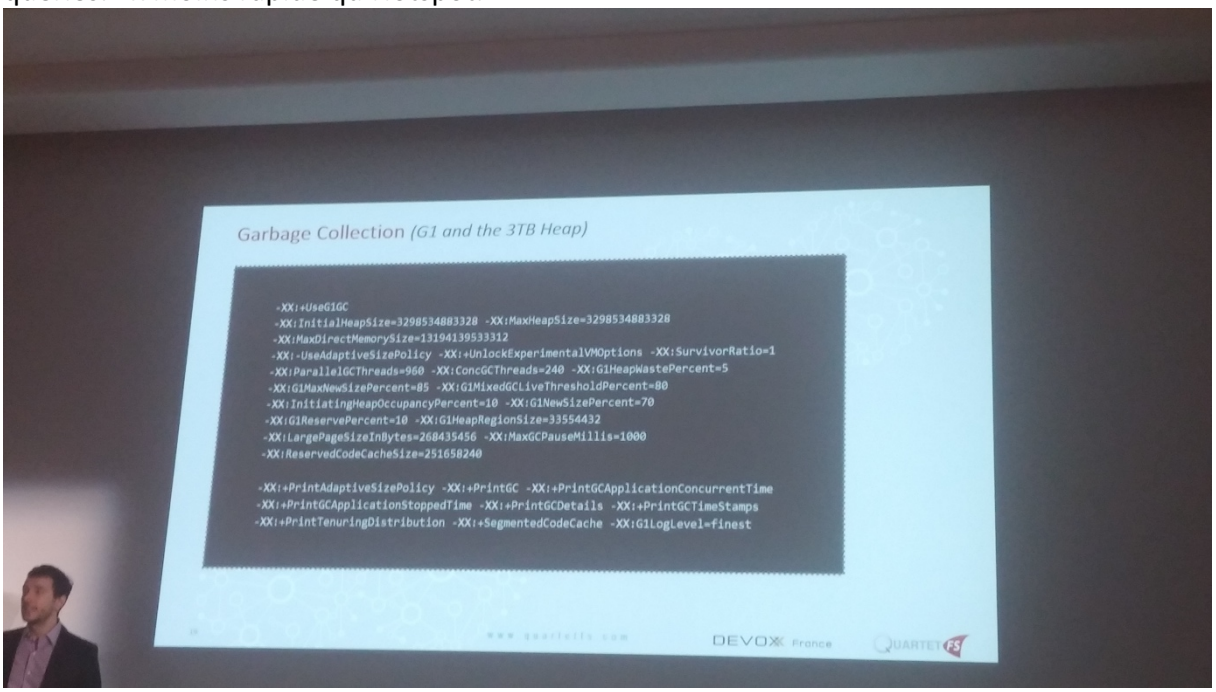
Le garbage collection et la heap de 3 To
Loin de la zone de confort de la JVM.
Rapprochement avec Oracle, IBM et Azul.
Travail collaboratif avec Oracle. Oracle a mise en place le matériel et ses ingénieurs.
ActivePivot est l’application de référence pour les gros volumes de données.

Stratégie :

- ⇨ Utiliser du GC G1 afin d’éviter le full GC stop the world

- ⇨ Eviter que les objets transients ne sortent de la new generation vers la old generation (promotion prématurée). Utilisation d'un gros SurvivorSpace (la moitié de la young generation)
- ⇨ Pour un nettoyage petit à petit : dès que 10% du Heap est occupé, le marquage se déclenche => marquage en continue
- ⇨ L'option prometteuse MaxGCPauseMillis n'a pas d'impact (peut être à l'avenir ?)

Fork de Hotspot, Zing est la JVM d'Azul qui a son propre algorithme de GC : le C4. C4 est pause less. Le prix de Zing => 10% de CPU. Cela a impact sur les temps de réponse des queries. 2x moins rapide qu'Hotspot.



Grâce au partenariat, Oracle améliore sa JVM. Ses améliorations seront disponibles dans Java 9.

Conclusion : Java peu désormais exploiter 16 To de mémoire.

<u>Questions</u> :

- ⇨ Paramétrage particulier de l'OS : le malloc par défaut n'est pas rapide. Utilisation de jemalloc sur Linux. Utilisation de pages de plusieurs Mo voir Go octets.
- ⇨ Dimensionnement de 3 To : ratio de 1 à 4.
- ⇨ L'augmentation du volume de données n'est-elle pas plus rapide que le matériel ? Limite du hardware ? Les Big memory servers se démocratisent depuis quelques années.
- ⇨ Azul essaie de diminuer l'occupation CPU
- ⇨ Une architecture distribuée ne serait-elle pas plus efficace et moins onéreuse ? Non à cause du réseau. Les prix des gros serveurs deviennent accessibles à cause de la concurrence.
- ⇨ Temps nécessaire pour charger les 16 To de mémoire ? 1 To en 3 ou 4 mn

- ⇨ Pourquoi ne pas avoir écrit ActivePivot en C ? Pour que la logique métier soit accessible aux 20 millions de dév Java.
- ⇨ Structures et collections utilisées : pas de collections Java. Tout a été refait en off-heap.
- ⇨ Outils utilisés : heap dump, et outils customs pour le off heap