Etude des bases de données Sous forme de Graphe dans le cadre du F.D.S.

RAPPORT DE STAGE JUIN-AOUT 2018

BOULET-GILLY Edmond | Employé à Thales Labège Services Labège | 31/08/2018

Le Stage en quelques mots

Contexte

Ce stage a eu pour but d'évaluer le marché actuel des bases de données sous forme de graphes en vue d'intégrer les données récupérées du Flight Dynamique System (F.D.S.) dans l'une d'entre elles.

Les données du F.D.S. sont liées entre elles par des relations tel que Parent-Child ou Reference. Elles sont massives et s'étendent sur des milliers de nœuds. Ainsi il est pertinent de les stocker dans des bases de données orientées graphe qui s'opposent aux bases de données relationnelle traditionnelle.

Les B.D.D. relationnelles et celles orientées graphe

Les Bases de Données Relationnelles

Les Base de données relationnelles sont celles dont les collection d'objets se font référence entre eux par des clés communes.

Par exemple, une base de donnée relationnelle peut contenir une table « *Employé* » qui possède un champ « *A_travaillé_pour* » qui a une valeur faisant référence au nom d'une entreprise qui pourra être retrouver dans la table « *Entreprise* ».

Le problème majeur de ce type de base de donnée étant la répétition de l'information et le fait de devoir parcourir une nouvelle table à la recherche du nom de l'entreprise si l'on souhaite récupérer des informations sur cette dernière. L'idéal serait d'avoir dans le champ « *A_travaillé_pour* » un pointeur qui indique directement l'information recherché vers la table « *Entreprise* ».

Les Bases de Données Orientées Graphes

C'est en effet le concept des bases de données orientées graphe. Chaque nœud (objet) peut posséder des champs aussi bien que des références à d'autres nœuds. Ces références sont représentées pas des liens (ou relations) qui ont des labels (« *A_travaillé_pour* » ici) et permettent d'accéder directement aux informations des nœuds voisins. Ainsi la récupération de l'information est accélérée et on parle réaliser des *transverses* ou de *parcourir le graphe*.

Le Marché Actuel

Rank Aug 2018	Rank Jul 2018	Rank Aug 2017	DBMS	Database Model	Score Aug 2018	Score Jul 2018	Score Aug 2017
1.	1.	1.	Neo4j	Graph DBMS	40.92	-0.95	+2.92
2.	2.	2.	Microsoft Azure Cosmos DB	Multi-model	19.52	+0.07	+10.10
3.	3.		Datastax Enterprise	Multi-model	7.30	-0.06	
4.	4.	↓ 3.	OrientDB	Multi-model	4.91	-0.48	-0.76
5.	5.	5.	ArangoDB	Multi-model	3.34	-0.11	+0.42
6.	6.	6.	Virtuoso	Multi-model	2.05	+0.06	+0.07
7.	7.	7.	Giraph	Graph DBMS	0.99	-0.01	-0.06
8.	8.		Amazon Neptune	Multi-model	0.81	+0.04	
9.	9.	↓ 8.	AllegroGraph	Multi-model	0.58	-0.01	-0.05
10.	↑ 12.	10.	GraphDB	Multi-model	0.57	+0.07	+0.00
11.	11.	↑ 16.	JanusGraph	Graph DBMS	0.54	+0.04	+0.31
12.	↓ 10.	↓ 9.	Stardog	Multi-model	0.53	+0.00	-0.04
13.	13.	↓ 11.	Sqrrl	Multi-model	0.35	-0.00	-0.16
14.	↑ 15.	↓ 13.	InfiniteGraph	Graph DBMS	0.30	+0.04	-0.00
15.	↑ 17.	15.	Blazegraph	Multi-model	0.28	+0.09	+0.02

Image 1 – Top 15 Base données proposant une orientation graphe Aout 2018
Source : https://db-engines.com/

Dans le cadre de l'étude, les bases de données jugées pertinentes par Thales, en prenant en compte la popularité, le support actuel du FDS, le modèle commercial et l'accessibilité du code source sont : Neo4J (leader mondial), *OrientDB* et JanusGraph.

Les moteurs tel que *Microsoft Azume Cosmos DB* et *Datastax Entreprise* n'étant pas Open Source, ils n'ont pas été sélectionnés. JanusGraph étant une préférence du groupe Thales. Il aurait été aussi pertinent, rétrospectivement, d'étudier ArangoDB.

Le modèle de Données du Flight Dynamic System

Les Données

Les données du F.D.S reposent sur 4 classes de nœuds et 3 classes de liens (ou relations)

Nœuds :

- *Folder* : possède un nom et un identifiant
- *Elément* : possède un nom, un identifiant, une version et des données scalaire, voir une liste de données scalaire
- *Data* : possède un nom, un identifiant, une valeur représentant un intervalle de temps et une longue série de valeur binaire (BLOB)
- *TData* : possède des données scalaires ou une liste de données scalaires
-

Liens :

- *Contain* : Traduit le fait qu'un objet contient un autre objet
- *Refer* : Traduit le fait qu'un objet fait référence à un autre objet
- *Lazy* : Traduit la nécessité d'exécuter une seconde requête en raison d'un très grand nombre de données sur le deuxième objet

Les Contraintes

Il y a aussi un certains nombres de contraintes qui régulent l'intégrité des données.

- Un *Folder* contient d'autres *Folder*s et/ou *Elements*.
- Un *Element* contient *Elements* et/ou *TDatas*.
- Un *Element* contient au moins un ou plusieurs *TDatas* s'il ne contient pas d'autres *Elements*.
- Un *Folder* peut contenir un autre *Folder* ou *Element* seulement si ce dernier ne le contient pas ou un de ses parents ne le contient pas.
- Un *Element* peut contenir un autre *Element* seulement si ce dernier ne le contient pas ou un de ses parents ne le contient pas.
- Un *Data*, *TData* ou *Element* est systématiquement contenu par un seul objet.
- Un *Element* peut faire référence à un ou plusieurs autres Elements.
- Un *Folder* peut être lier à un autre *Folder* par un lien *Lazy*, ce dernier contient alors systématiquement de nombreux *Datas*.
- Un *Folder* reçoit systématiquement un lien *Contain* ou un lien *Lazy*.

Les Tests à Effectuer

Le principal aspect de ces moteurs de bases de données à tester dans le cadre du F.D.S. est leur capacité à monter en charge et vérifier si elles maintiennent des performances convenables lorsqu'elles contiennent une grande quantité de données.

Ainsi les paramètres testés tel que la vitesse d'exécutions, la capacité à récupérer les données, la fiabilité et l'empreinte mémoire sur le disque ont été évalué à différentes tailles de graphe de manière à pouvoir déceler des courbes de tendances dans les graphiques.

Les tests ont été effectuées à travers différentes Architecture (Single Instance et Architecture distribuée) et à travers différentes implémentations (Single Thread et Multithread)

Le Support Matériel

Les tests de Single Instance ont été effectué sur une machine virtuelle lancé sur ordinateur portable disposant de la configuration suivante :

- Linux – Red Hat x64-bit
- 2 cœurs sur 8 d'un Intel i7-7820HQ 2.90 GHz
- 8192 Mo de RAM

Les tests en architecture distribuée ont été réalisée sur un cluster de 3 machines fixes sans interface graphique ayant pour configuration :

- CentOS Linux release 7.4.1708 (Core)
- Intel(R) Xeon(R) CPU 4 cœur E31225 @ 3.10GHz
- 4 * 2GB of RAM at 1333Mz DDR3

OrientDB

A PROPOS

OrientDB est la base de donnée NoSQL Multi-model la plus polyvalente parmi celle étudiée ici.

La force d'*OrientDB* réside dans la non-contrainte du format des données que le moteur peut gérer. Les objets peuvent être des documents, des graphiques ou des classes personnalisées, il sera possible sous *OrientDB* de les manipuler des de créer des liens pour rendre la base de données orientée graphe. A l'heure actuelle *OrientDB* est constituée de 2 fois moins de code que Neo4J et son modèle économique est plus séduisant dans la mesure seulement quelques fonctionnalités d'administration de cluster sont bloquées derrière l'édition *enterprise*. Le code source *d'OrientDB* étant sous License Apache 2.

Les tests ont été réalisés sous *OrientDB* 3.0.0 pour le single instance et 3.0.5 pour l'architecture distribuée. Dans cette version l'API de prédilection est la Multi-Model-API, et l'architecture distribuée recommandée par la documentation et implémentée de base dans *OrientDB* est de type Haute Disponibilité. Cependant le F.D.S., dans son état actuel, repose sur l'API *Tinkerpop Gemlin*. Cette API permet aussi d'interagir avec *OrientDB*, pour cette raison les 2 APIs ont été testés.

SINGLE INSTANCE

TinkerPop API

L'API *Tinkerpop* permet d'interagir avec *OrientDB* à travers des transactions réalisées, dans notre choix d'implémentation, par du code Java. Pour chaque objet/nœud que l'on veut créer, il faut l'initialiser localement dans la mémoire de notre applications clients puis pousser la transaction pour le que le server *OrientDB* effectue à son tour la transaction sur la base de donnée.

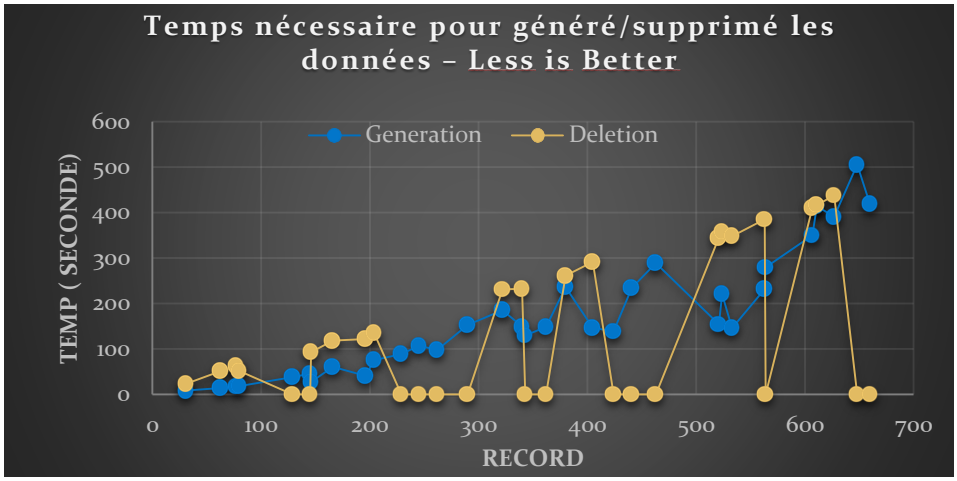

Image 2 – Performance sous *OrientDB* avec *Tinkerpop* API

Comme on peut voir sur l'image 2, les performances de l'API *Tinkerpop* sont assez décevantes avec 500 secondes pour créer 650 nœuds contrairement au millier de nœud par seconde annoncé par *OrientDB*. De plus les points où la suppression des données a pris 0 secondes représentent en réalité des moments où l'exécution de l'application JAVA a renvoyé une erreur car la transaction n'a pas pu être réalisé, peut-être à cause d'un problème de synchronisation des données.

Après en avoir parler sur le git *OrientDB* (ticket #8349 sur le *Git* officiel), il semblerait que *Tinkerpop* ne soit plus un outil de prédilection pour interagir avec *OrientDB* et qu'il faudrait privilégier l'API Multi-Model (MMAPI).

MMAPI

L'API Multi-Model consiste à traiter n'importe qu'elle base de donnée comme étant une base de donnée document mais avec laquelle il est possible de mettre des liens entre chaque objet et obtenir de ce fait une base de donnée orientée graphe.

L'API Multi-Model est directement développé par *OrientDB* et va permettre de communiquer avec la base de donnée aussi bien en effectuant des transactions qu'en exécutant directement de requêtes SQL dans le langage d'*OrientDB*.

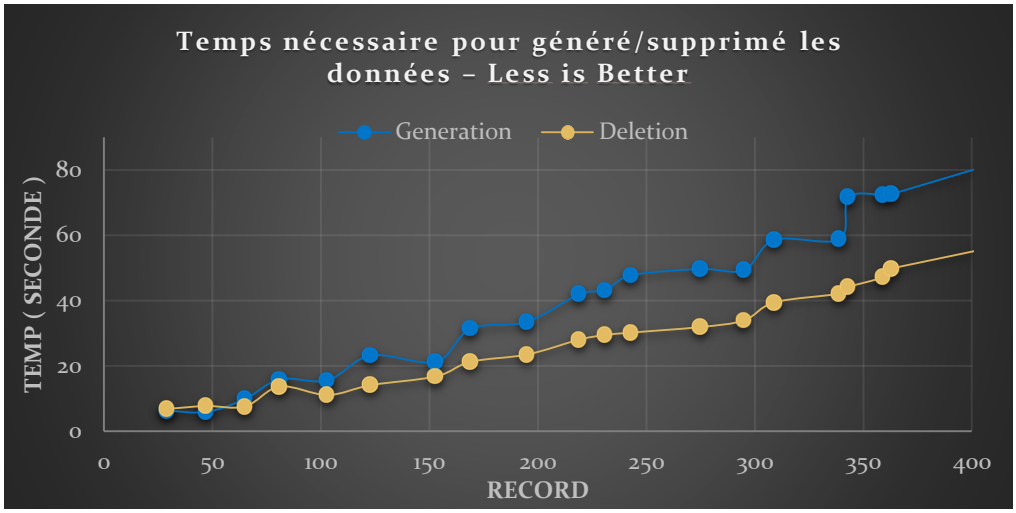

Image 3 – Performance sous *OrientDB* avec MMAPI

Avec l'API Multi-Model on peut observer de bien meilleur performance et une stabilité accrue.

Comme les données sont basées sur le modèle du F.D.S., la génération du graphe a été conçu pour chaque nœud fasse environ 1 Mo, de ce fait une base de donnée d'environ 350 nœuds fait environ 350 Mo, plus les logs, plus le backup qui donne la possibilité de roll-back et ainsi pouvoir annuler une transaction.

Le problème étant que pour tester la capacité d'un moteur de base de donnée à garder son intégrité sans perdre en performance lors de la monté en charge est qu'une base de donnée de mille nœuds prendraient 1 Go et un million de nœuds 1 To. De plus laisser, au moteur, le temps d'écrire sur le disque serait conséquent et contreproductif. C'est pour cette raison que par la suite des tests ont été réalisé en enlevant la majorité des données de chaque nœud mais en conservant l'architecture général du F.D.S.

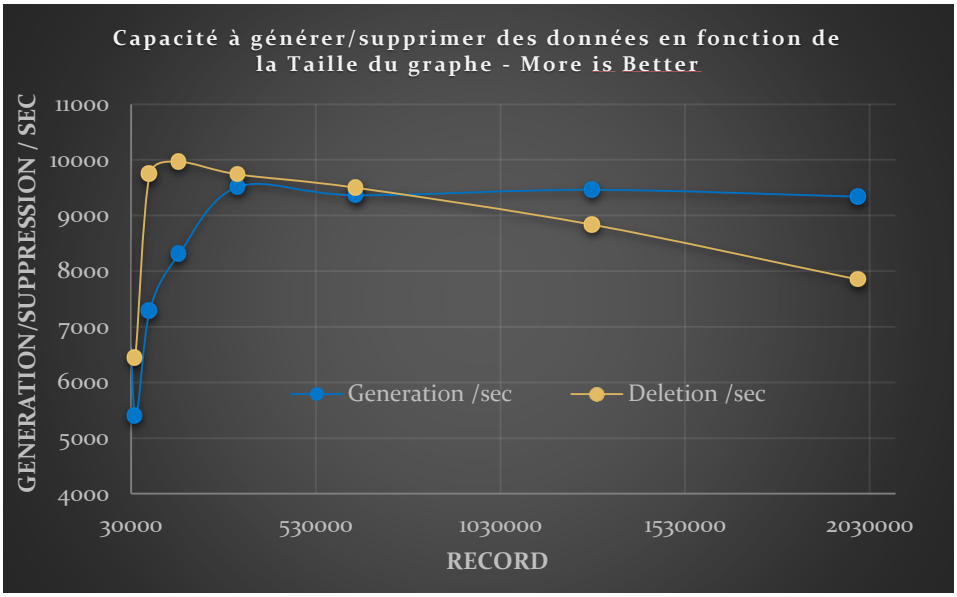

Image 4 – Performance sous OrientDB MMAPI sans données avec MMAPI

Avec ce test on peut observer la stabilité de la base donnée, même à plusieurs millions de nœuds, de hautes performances avec une dizaine de milliers d'actions par secondes et une légère décroissance des performances avec l'augmentation de la taille du graphe.

ARCHITECTURE DISTRIBUEE

De type Haute disponibilité

OrientDB implémente l'architecture distribuée de type haute disponibilité. Chaque membre du cluster va avoir le droit d'effectuer et de traiter des transactions tout en propageant la transaction effectuée aux autres membres du cluster.

Chaque transaction implémente le principe le *Core Majority*, c'est-à-dire qu'une transaction est considérée comme faite (notification de l'utilisateur que la transaction est effectuée) dès le moment où plus de la moitié des membres du cluster ont effectué la transaction.

Tel que *OrientDB* est livré, le mode distribué consiste à ce que tous les membres aient, à tout moment, la même base de donnée stockée localement. De manière à ce que chaque serveur puisse répondre à des requêtes *read* indépendamment et ainsi avoir de meilleur performance en lecture par rapport au mode Single Instance.

Il est aussi possible d'indiquer qu'un objet ou une collection d'un objet se situent uniquement dans un membre du cluster précis. C'est utile si l'on sait qu'un client va systématiquement se connecter à un tel membre pour demander uniquement

des informations liées à cet objet. Il reste possible de créer des relations entre l'objet concaténer au membre du cluster et les autres objets distribuées dans le cluster.

Dans notre étude, l'architecture basique de réplique des données a été testée.

Déploiement et communication entre Instance

Le déploiement dans notre étude s'est effectué par *Ansible*, en copiant les packages d'*OrientDB* sur les différentes machines du cluster puis en exécutant une instance de /bin/dserver.sh sur chaque machine.

Ce script demander un mot de passe root puis un nom de nœud du cluster qui peut être automatiquement générer puis, l'instance du serveur attend qu'un nombre minimum de machine rejoigne le cluster.

Ce nombre minimum de machine dépend des paramètres entrés dans fichier de configuration au même titre que du nombre minimum de machine en temps de marche (3 et 3 par défaut).

Si le réseau sur lequel les machines se situent ne supporte pas les paquets IP multicast (ce qui est fréquent) il faut préciser directement les différentes adresses IP des machines du cluster dans le */conf/hazelcast.xml* d'*OrientDB*.

Une fois que les machines se sont repérés entre elles, elles entament une période de synchronisation puis sont disposées à répondre aux requêtes.

Single Thread

Les tests en Single-Thread ont été effectué sur les même bases que les tests en Single Instance, un script java déployé localement sur une des machines du cluster effectuaient des transactions avec cette machine uniquement, les transactions étant transmises par la machine aux autres membres du cluster.

Multi Thread

Les tests en Multi-Thread ont été effectué en séparant en 3 Threads le travail effectué en Single-Thread et en divisant leur travail effectué par 3. Chaque thread se connecte à un serveur différent pour effectuer ses transactions.

Les Threads sont créés après la génération du dossier *root*, elles se rejoignent après la génération des nœuds, se séparent pour faire faire des récupérations de données, se rejoignent et se séparent pour supprimer un à un les nœuds du graphe

De cette manière la charge de travail est uniformément réparti sur les trois serveurs. Comme la génération des données se fait en replissant le *Root Folder* de nœuds qui consistent des branches indépendantes, il n'y a pas de conflit de requête.

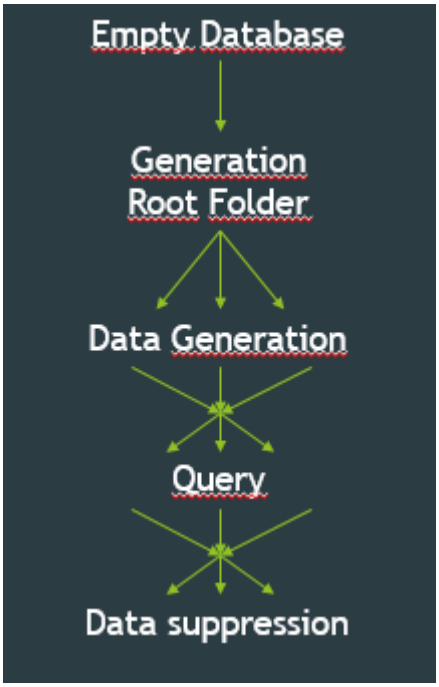

Image 5 – Cheminement de l'algorithme de test

Ob obtient ces résultats :

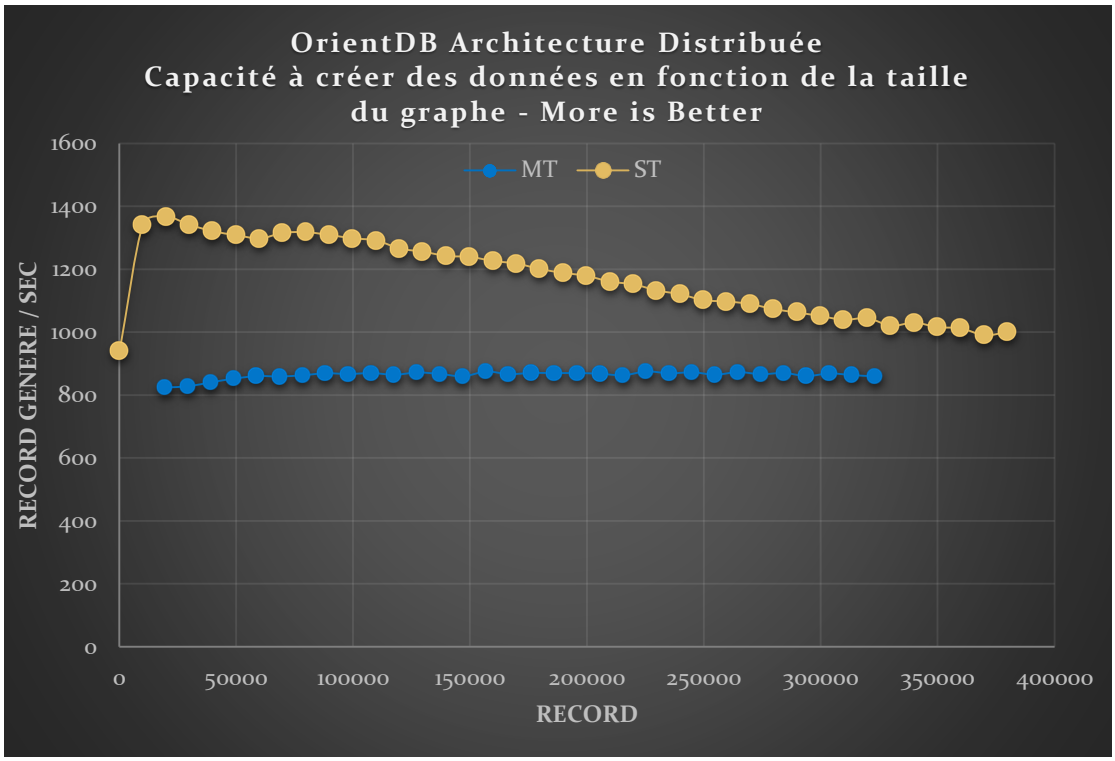

Image 6 – Performance sous *OrientDB* Architecture Distribuée en génération

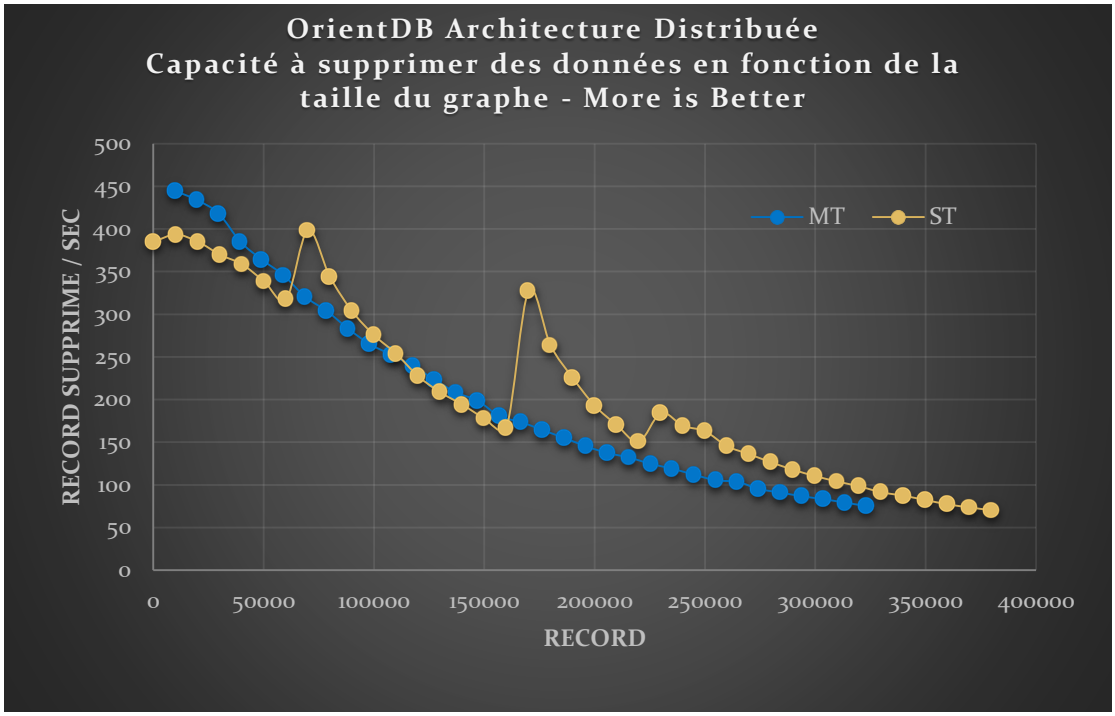

Image 7 – Performance sous *OrientDB* Architecture Distribuée en suppression

On peut observer une légère décroissance des performances lorsque le taille du graphe augmente dans la majorité des tailles. Il est normal que les performances diminuent en architecture distribuées en raison du temps nécessaire pour que les serveurs se synchronisent entre eux après chaque transaction.

On remarque aussi une hausse de performance régulière quand il s'agit de supprimer le contenu de la base de donnée. C'est dû au fait que les tests n'ont pas pu être effectuer d'un seul coup pour des raisons de restrictions matérielles, de ce fait une remise à neuf régulière de la base était nécessaire. On observe donc qu'il y a une hausse de performance quand il s'agit de commencer sur une instance neuve.

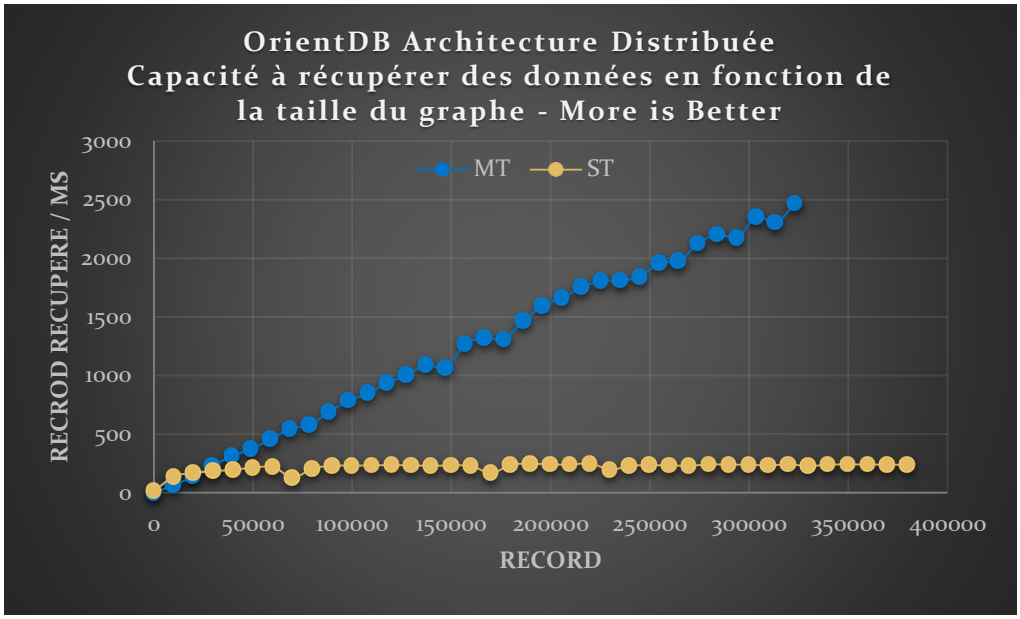

Image 8 – Performance sous *OrientDB* Architecture Distribuée en lecture

On observe par contre que la parallélisassions des instances d'*OrientDB* augmentent grandement la disponibilité des données en lecture.

Transaction groupée

Jusqu'à maintenant la génération et suppression des données étaient réalisées de manière à ce que chaque transaction apporte le minimum de changement à la base de donnée, de manière à quantifier le temps nécessaire à la transaction élémentaire. Par exemple pour la génération chaque nœud était créé par une transaction puis les nœuds sont supprimés un à un.

Il a aussi été testé des modes de génération et suppression par paquet de nœuds de manière à vérifier si l'on était limité par le nombre de transaction ou bien par la vitesse d'écriture (ces tests seront plus pertinents lorsque l'on passera sur Neo4J).

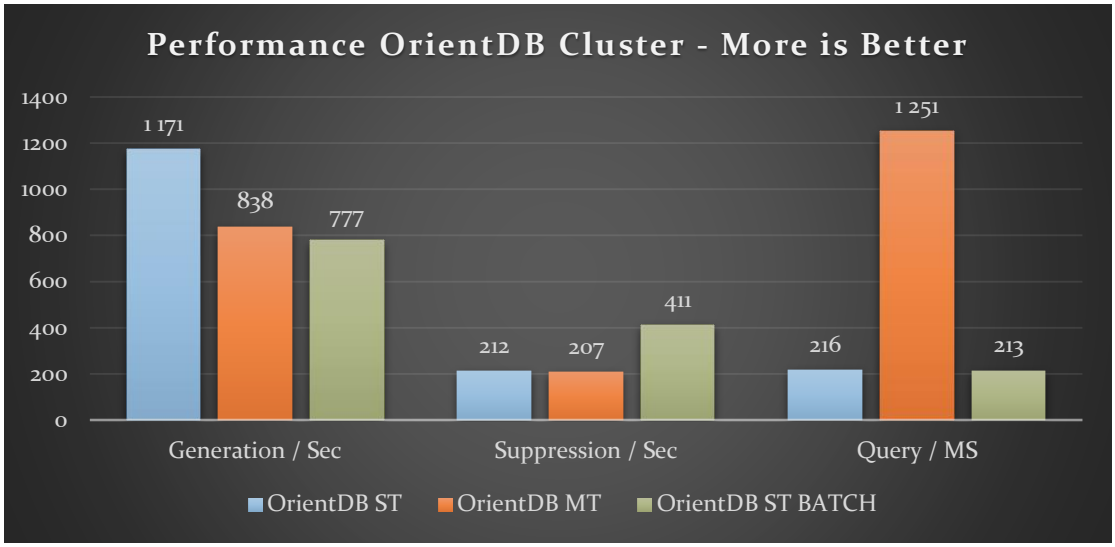

Image 9 – Performance sous *OrientDB* Architecture Distribuée

Quand on voit le graphique ci-dessus on peut voir que la taille des transactions n'a pas beaucoup d'importance, les baisses de performances vis-à-vis du Single-Thread non BATCH sont due au fait que le batch permet d'arriver à une quantité de nœud plus grande, plus rapidement, ce qui occasionne une basse de performance. Par contre supprimer est 2 fois plus rapides en mode BATCH.

On remarquera sur ce graphique l'absence de test Multi-Thread BATCH, la raison pour laquelle ces tests n'apparaissent et parce que les résultats n'étaient que très peu pertinents et ne faisaient que traduire la faiblesse d'*OrientDB* à gérer des transactions concurrentes de grande taille. Pour comprendre ce qu'il se passe, il faut s'intéresser à la façon dont *OrientDB* gère les transactions.

Supposons un cluster de 5 machines, quand le client transmet une transaction à la première machine, l'instance exécute la transaction mais ne notifie pas tout de suite le client du succès de la transaction, pour cela il doit s'assurer que plus de la moitié des instances ont effectué la transaction. A ce stade il n'y a 1/5 machine qui l'ait effectuée. La première machine transmet la transaction à une deuxième machine qui l'effectue : 2/5. Mais arrivée à la troisième machine supposons qu'une autre transaction a eu lieu et cette dernière n'est pas compatible avec la première. L'incompatibilité peut venir de la modification de la même donnée ou que tout simplement la version de la base de donnée est différente. Les 2 machines doivent annuler la transaction en cours et notifier le client en lui envoyant une erreur *OConcurrentTransactionException*. Comme ce problème arrive surtout quand les transactions sont conséquentes et prennent du temps à se mettre en place, il est normal que l'erreur arrive systématiquement en Multi-Thread BATCH.

La manière dont *OrientDB* Technologies recommande d'effectuer ce genre de transaction est par le procès de *transaction optimiste*. L'utilisateur doit refaire la transaction tant qu'il reçoit un message d'erreur.

Implémenter ce genre de transaction pour générer est possible, il suffit de mettre des boucles dans chaque thread qui sont composés de la *transaction*, d'un *break*, et d'un *catch* dans cet ordre. De cette manière la boucle continue seulement si l'exception a été attrapée.

Le problème avec cet algorithme fut la suppression de nœud. A cause des mécanismes de *roll-back*, de verrouillage de données et transaction concourantes, le fait de lancer 3 threads qui suppriment les nœuds par parquet a tendance à bloquer la base de données et même quand l'application client a arrêté de fonctionner, les données restent verrouiller laissant la base de donnée inutilisable.

Neo4J

A PROPOS

Neo4J est similaire sur le plan fonctionnel à *OrientDB*, disposant d'une installation unifiée dans une archive, un application web une fois que le serveur est lancé et des possibilités d'interagir en Java en faisant des transactions à travers une API.

Les différences principales étant que l'utilisateur sera amené à faire du *Cypher*, langage développé spécialement par *Neo4J* pour *Neo4J* au lieu que du SQL, qu'on ne peut avoir qu'une une seule base de donnée par instance et que le modèle économique est moins *utilisateur friendly* qu'*OrientDB*. En plus de ça, *Neo4J* n'est pas aussi polyvalent qu'*OrientDB* dans la mesure où Neo4J est un moteur purement base de donnée orientée graphe et qu'il supporte moins de type d'objets différents.

Les tests seront réalisés sur *Neo4J* 3.4.0, avec l'API driver JAVA, combinant exécution de requêtes *Cypher* et transactions.

SINGLE INSTANCE

En single instance, *Neo4J* a une configuration similaire à *OrientDB*, les tests ont été effectué avec une instance *Neo4J* exécutée localement sur la machines avec une application JAVA exécutée aussi localement et qui se connectait au server et envoyait des requêtes et des transactions.

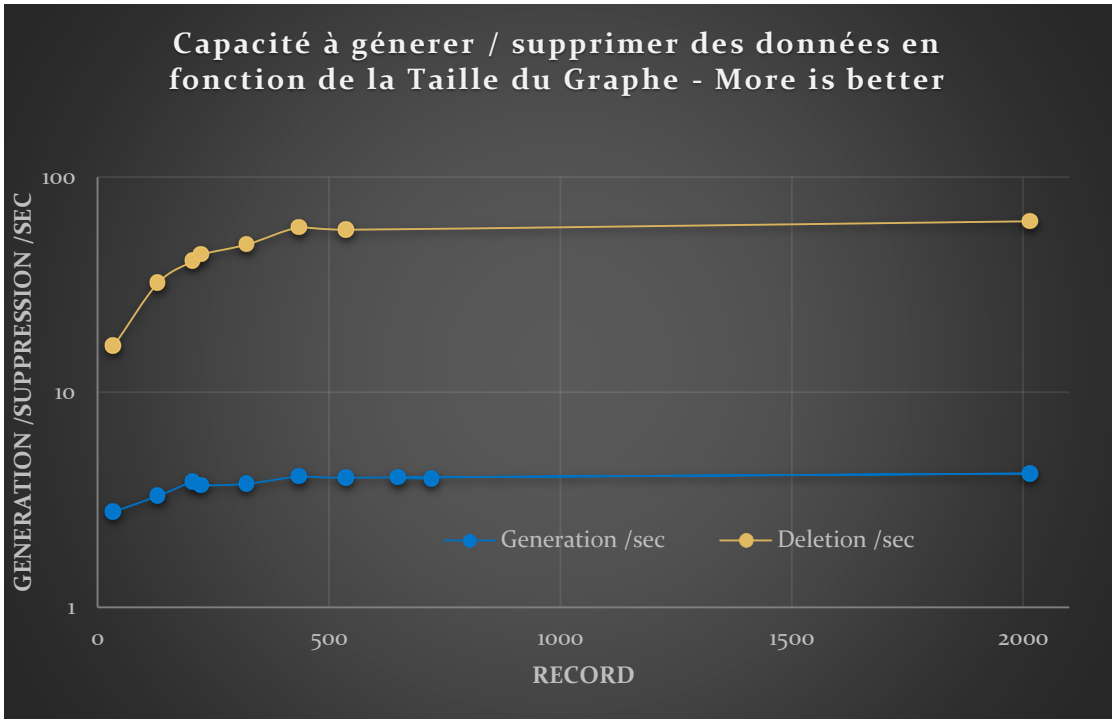

Image 10 – Performance sous *Neo4J* S.I. avec données

Encore une fois avoir des nœuds de 1 Mo ralentit fortement le processus et limite notre capacité à monter en charge dans des plages de temps réalisable.

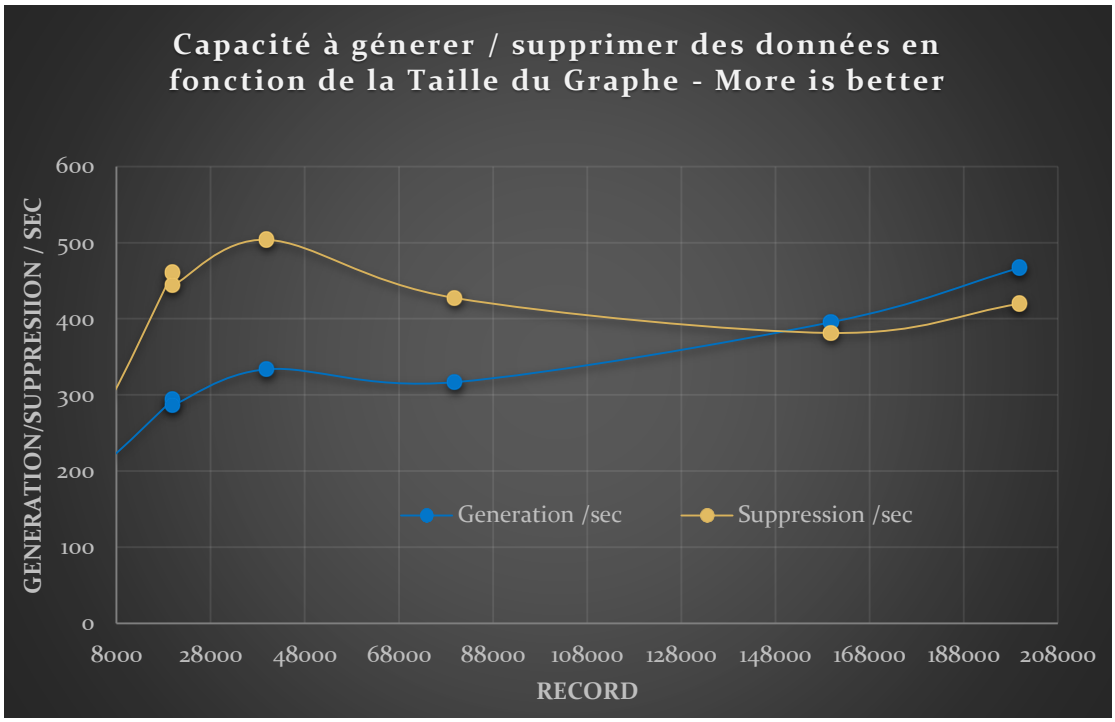

Image 11 – Performance sous *Neo4J* S.I. sans données

On peut observer des performances moins élevées qu'*OrientDB* mais, ces chiffres restent quand même intéressant quand on considère que chaque transaction ne fait que créé un seul nœud/relation. Comme on le verra par la suite il est possible d'obtenir de meilleur performance sous *Neo4J* en regroupant les actions sous une seule transaction et ainsi en faire plus en une fois et en moins de temps.

ARCHITECTURE DISTRIBUEE

Architecture Causal

Avant toute chose, l'architecture Haute Disponibilité chez *Neo4J* fut présente et mise en avant dans les versions antérieures à celle testée ici. Cette architecture, bien que mettant en avant la disponibilité en terme de ressource effectue un compromis sur l'intégrité de la base de donnée comme vu sur *OrientDB*. Aujourd'hui cette architecture est considérée comme *legacy* chez Neo4J bien que toujours présente dans les possibilités de configurations. L'architecture conseillée aujourd'hui est dite *Causal*.

L'architecture *Causal* fut donc l'architecture testée dans cette étude. Dans cette architecture, vos instances Neo4J sont de type *Replica* ou bien *Core*.

Les types *Replica* étant des instances qui ne font que consulter les *Cores* en lecture et n'ont pas le droit d'écrire sur la base de donnée, elles sont optionnelles et ne sont là que pour décupler la capacité en lecture de la base de donnée.

Les types *Cores* sont les instances qui vont avoir le droit d'écrire et de lire sur la base de donnée. Quand un client envoie une requête d'écriture, il va l'envoyer à une de ses machines. Cela dit une machine *Core* n'a pas le systématiquement le droit de traiter les requêtes d'écriture. Parmi les machines *Core* il y a un *Leader*, élu en fonction des performances et les autres *Cores* sont *Follower*. Toutes les requêtes sont traitées par la machines *Leader* dont le travail est simplement de vérifier l'intégrité de la requête sur la base de donnée avant de la transmettre aux autres *Cores* pour que la transaction soit effectuée sous le principe de *Core Marjority*.

Un client n'est pas obligé d'envoyer directement sa requête à la machine Leader. En soit comme il n'est pas censé avoir accès à cette information il est normal qu'il ait la possibilité de l'envoyer à n'importe quelle machine *Core* et que sa requête soir routé vers la machine *Leader*. Le routage s'effectue par le protocole *bolt*. Lorsque le client envoie sa requête, l'adresse de la machine de destination est de la forme *http+routing://192.168.1.1/*, le *+routing* va avoir comme effet que la machine *Core* qui reçoit la requête va la transmettre à la machine *Leader*.

Neo4J implémente aussi de nombreux mécanisme de réélection de *Leader* en cas de chute de ce dernier de manière à garantir le fonctionnement du cluster. On retrouve des paramètres que l'utilisateur doit indiquer tel que le nombre de machine nécessaire au déploiement du cluster avant la mise en route ou encore le nombre de machine at *runtime*.

Déploiement et communication

De manière similaire au déploiement d'*OrientDB*, le déploiement se fait en précisant sur chaque machine l'adresse IP des autres et en exécutant Neo4J sous forme de service sur chaque machine du Cluster.

Le déploiement dans notre étude s'est effectué par Ansible, en copiant les packages *Neo4J* sur les différentes machines du cluster. Une fois le service lancé et que les machines ont communiqué entre elles, une application JAVA a été déployée sur une d'entre elles et a effectué les tests.

Single thread

En Single-Thread l'étude a été réalisée dans le cas plus commun, c'est-à-dire le cas où le client contacte un *Core* qui n'est pas Leader, de ce fait chaque transaction est routée avant d'être traitée.

Multi thread

En Multi-Thread, 3 Threads sont créés de manière similaire à *OrientDB* et se connectent chacune à une machine différente, de ce fait seulement 2/3 des transactions sont routés vers le Leader.

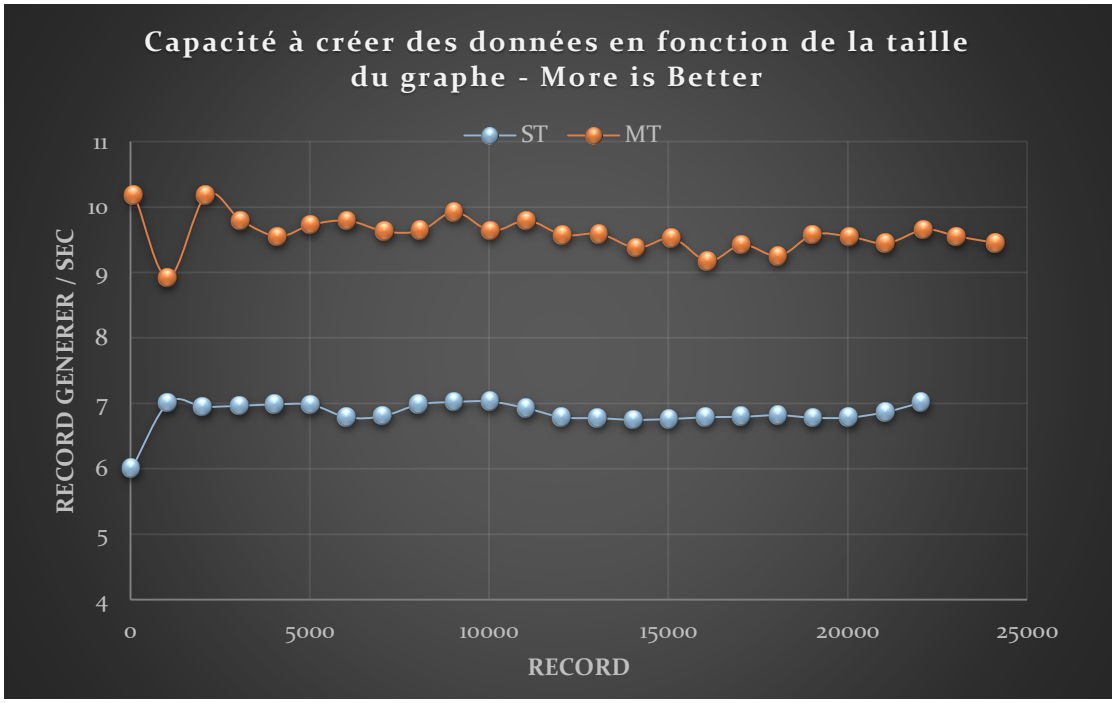

Image 12 – Performance sous *Neo4J* Architecture Causal ST et MT

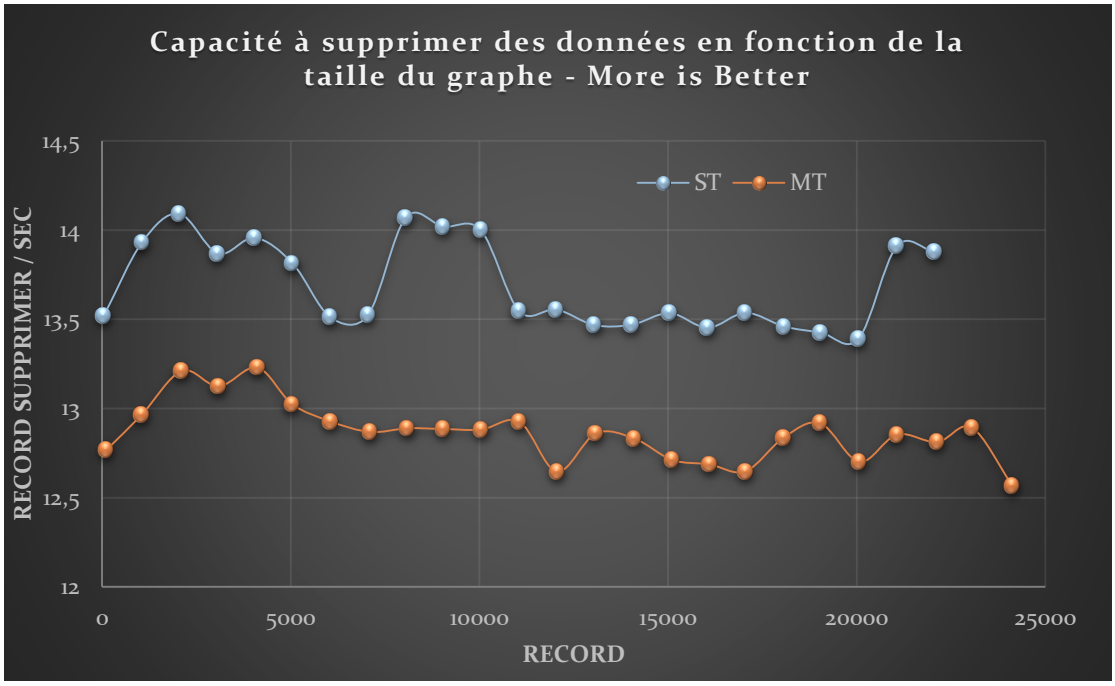

Image 13 – Performance sous *Neo4J* Architecture *Causal* ST et MT

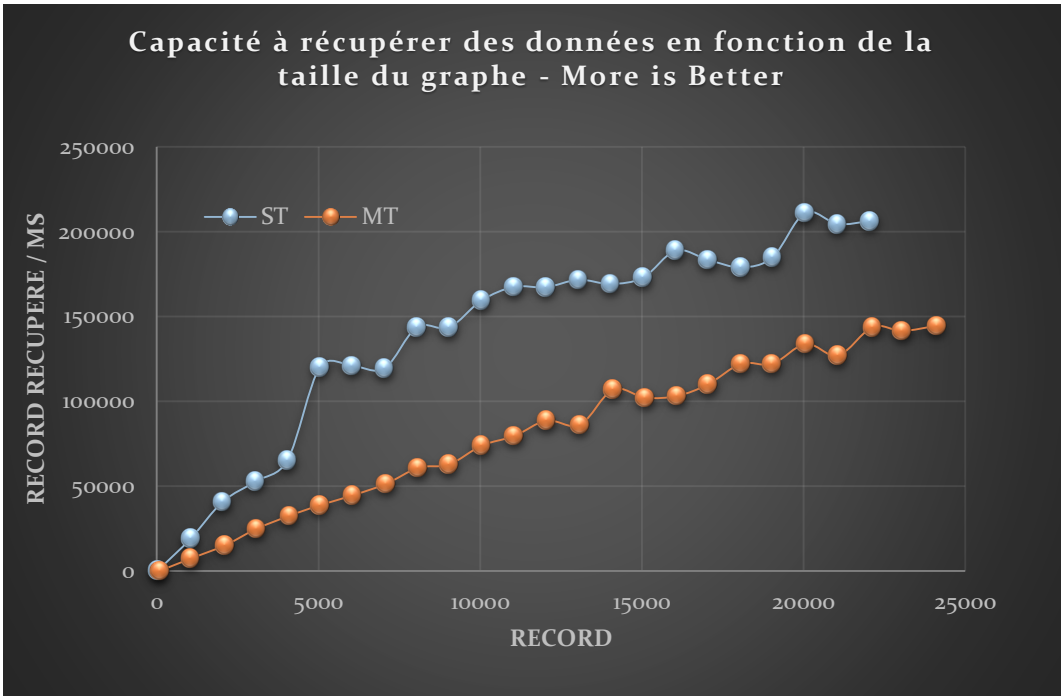

Image 14 – Performance sous *Neo4J* Architecture Causal ST et MT

On peut voir sur les graphiques que les transactions sont très lentes et on atteint difficilement la vingtaine de nœud par seconde.

Par contre en lecture, il faut savoir que chaque Thread en mode Multi-Thread avait pour objectif de lire 1/3 du Graphe de manière à avoir un résultat similaire au

travail réalisé par le Single-Thread qui récupérait le graphe en entier. L'architecture distribuée apporte de bonne performance en écriture quand il s'agit de faire du Multi-Thread.

BATCH

Des tests ont ensuite effectuée en réalisant des transactions groupés en utilisant des procédures effectuant des commit périodiques tout au long de la transaction. La bibliothèque de fonction utilisée est la bibliothèque de fonction *Apoc*, qui peut être trouvé ici.

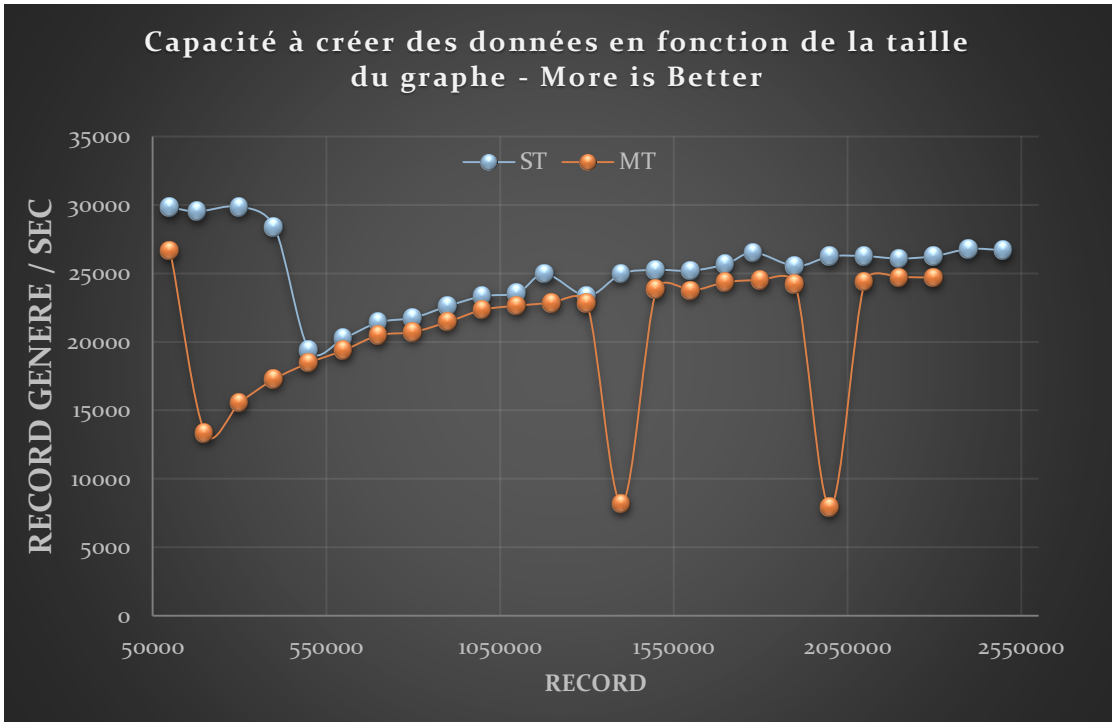

Image 15 – Performance sous *Neo4J* Architecture Causal ST et MT BATCH

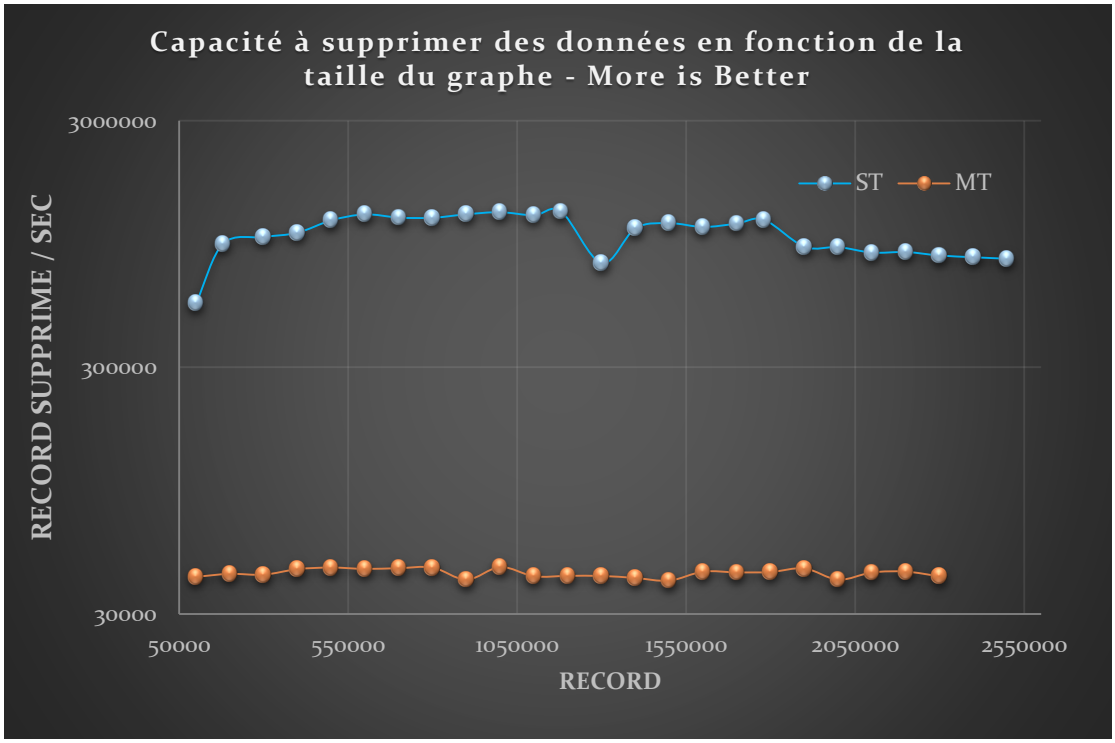

Image 16 – Performance sous *Neo4J* Architecture Causal ST et MT BATCH

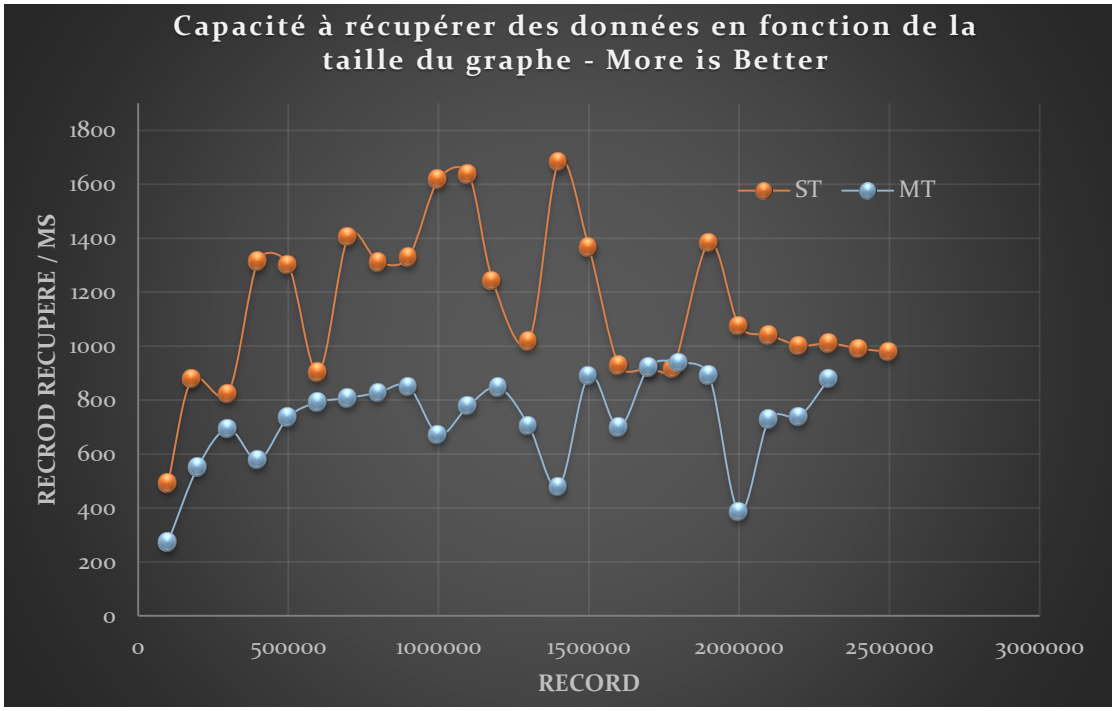

Image 17 – Performance sous *Neo4J* Architecture Causal ST et MT BATCH

On peut voir le Neo4J était limité au nombre de transaction et non pas en écriture. On observe cependant une baisse de performance quand il s'agit de supprimer les données en Multi-Thread. Il semblerait que les problèmes d'intégrité soient moins présents cela dit environ 43 000 nœuds par seconde reste très intéressant.

Si on met en parallèle les résultats en Architecture distribuée on obtient en moyenne les résultats suivants :

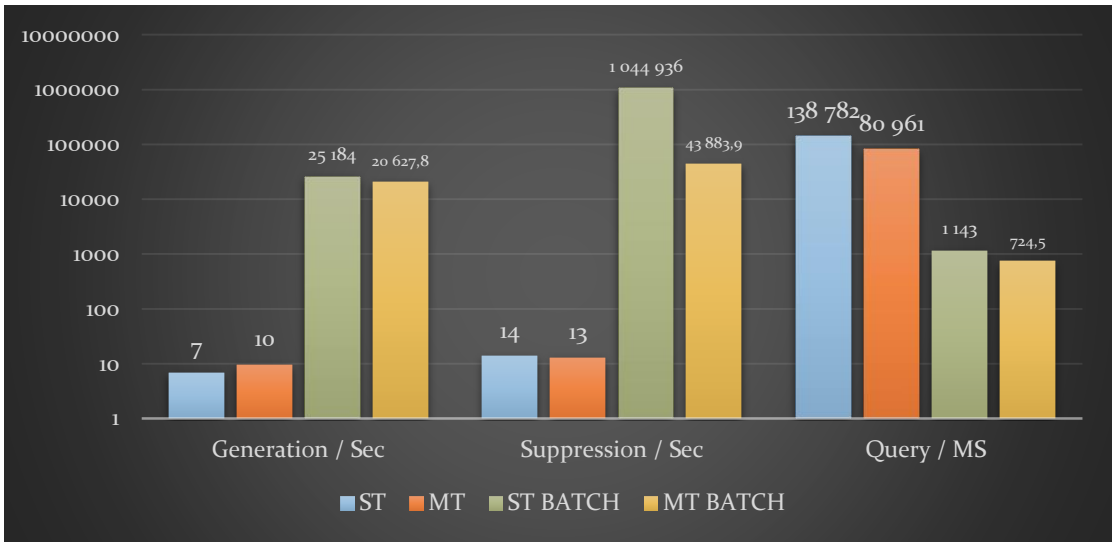

Image 18 – Performance sous *Neo4J* Architecture *Causal*

Il faut garder en mémoire que les résultats des colonnes *Query/MS* ont été obtenu pour un nombre de nœud faible en transaction unitaire et un nombre de nœud élevé, la décroissance de performance est dû au fait que les performances sont toujours plus intéressantes lorsque le base de donnée est moins chargée.

JanusGraph

A PROPOS

JanusGraph a la particularité de ne pas avoir de *front-end* et *back-end* développé directement pour *JanusGraph*. En *front-end* il est possible d'utiliser *Tinkerpop* et en *back-end* il est possible d'utiliser *Apache Cassandra*, *HBase* ou encore *BerkeleyDb*. Ce qui fait de JanusGraph un moteur de base de donnée très modulable, en contrepartie il demande plus de configuration et de travail de la part de l'utilisateur par rapport à *Neo4J* ou *OrientDB* dont la structure est plus unifiée par les développeurs.

Un autre désavantage est le statut de développement de *JanusGraph*, en version 0.2.0 seulement, ce moteur de base de donnée ne dispose pas des même fonctionnalités que les autres vu précédemment (pas d'interface web, pas de semi-SQL).

Ici *JanusGraph* a été testé avec l'API *TinkerPop* et *CassandraDB* en back-end

SINGLE INSTANCE

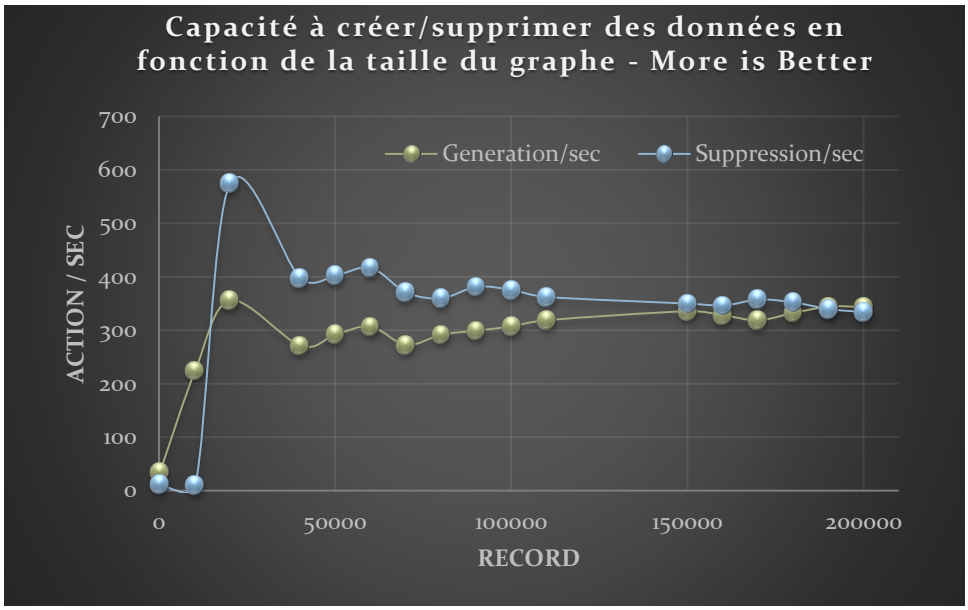

Image 19 – Performance sous *JanusGraph* Single Instance

Les tests en Single Instance montre de bonne performances, malheureusement les tests n'ont pas pu être continuer à plus de 200000 nœuds à cause de l'instabilité de *JanusGraph* qui a aussi montré des instabilités lors de tester les requêtes. Seulement quelques valeurs ont été récupérés et sont affichés sur l'image 22.

A vue de ces faiblis performances, *JanusGraph* n'a pas été testé en Architecture distribuée.

Alignement des résultats

SINGLE INSTANCE

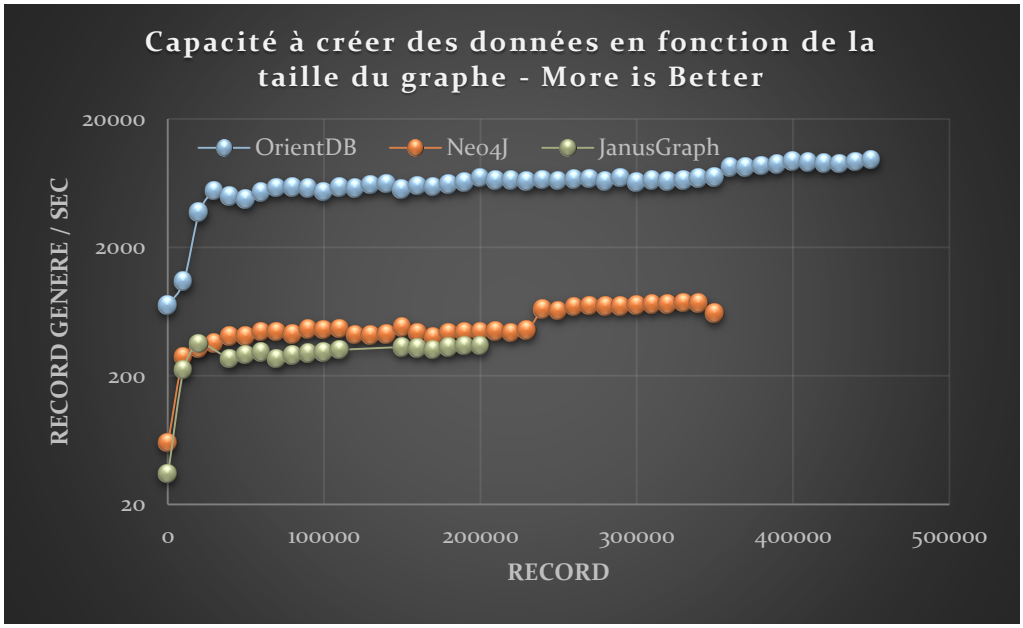

Image 20 – Performance en génération Single Instance

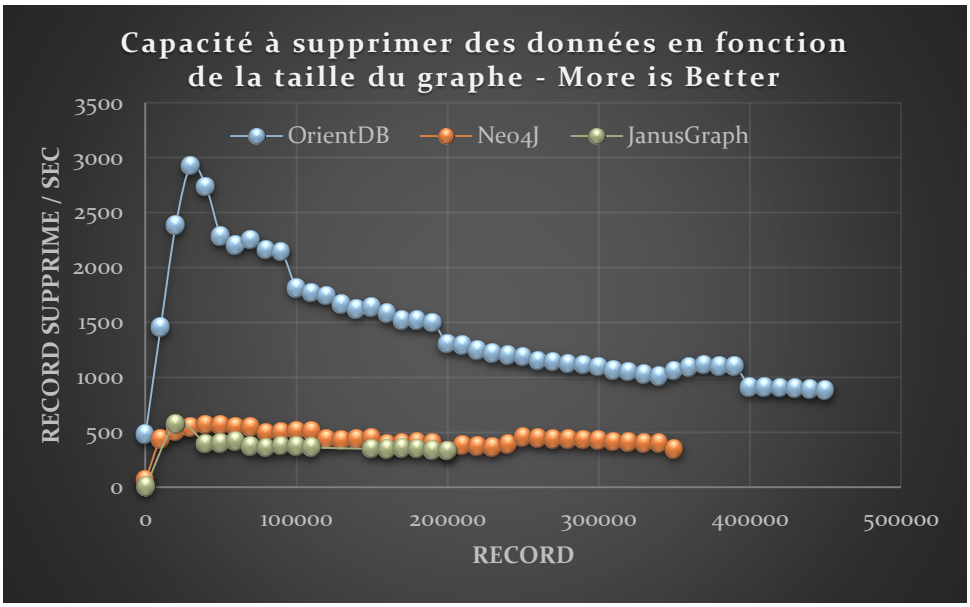

Image 21 – Performance en suppression Single Instance

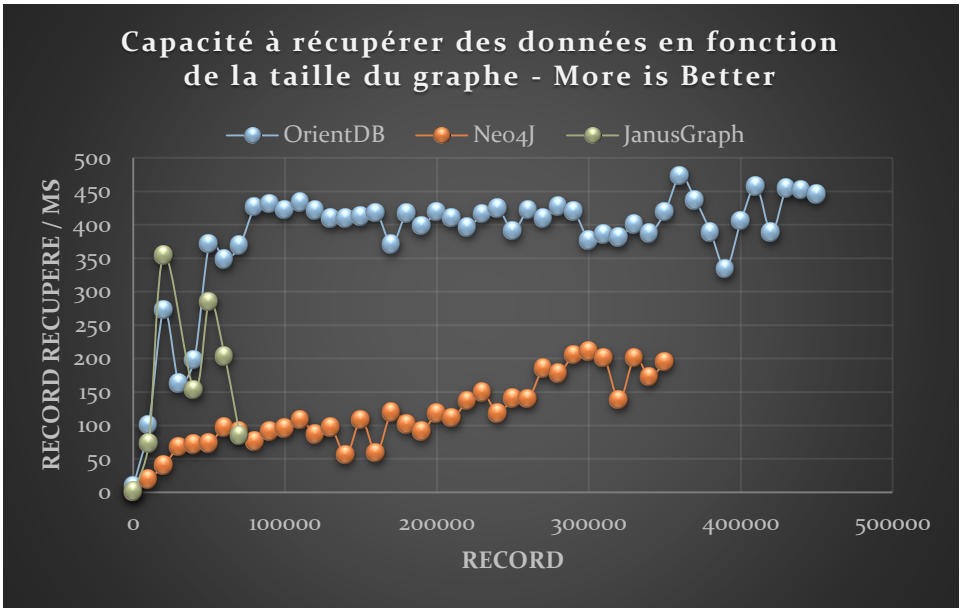

Image 22 – Performance en récupération de données Single Instance

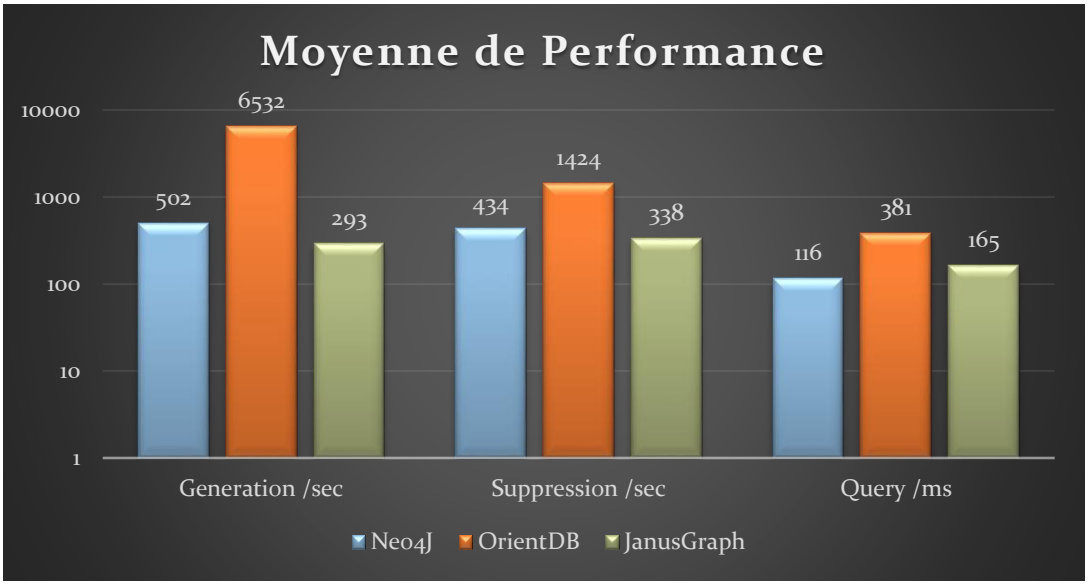

Image 23 – Performance en moyenne Single Instance

Comme on peut le voir les tests réaliser avec *JanusGraph* n'ont pas menés à grand-chose comparé à *OrientDB* et *Neo4J*.

On observe qu'*OrientDB* est devant niveau performance mais avec un déclin net quand il s'agit de supprimer les données quand la taille du graphe augmente.

ARCHITECTURE DISTRIBUEE

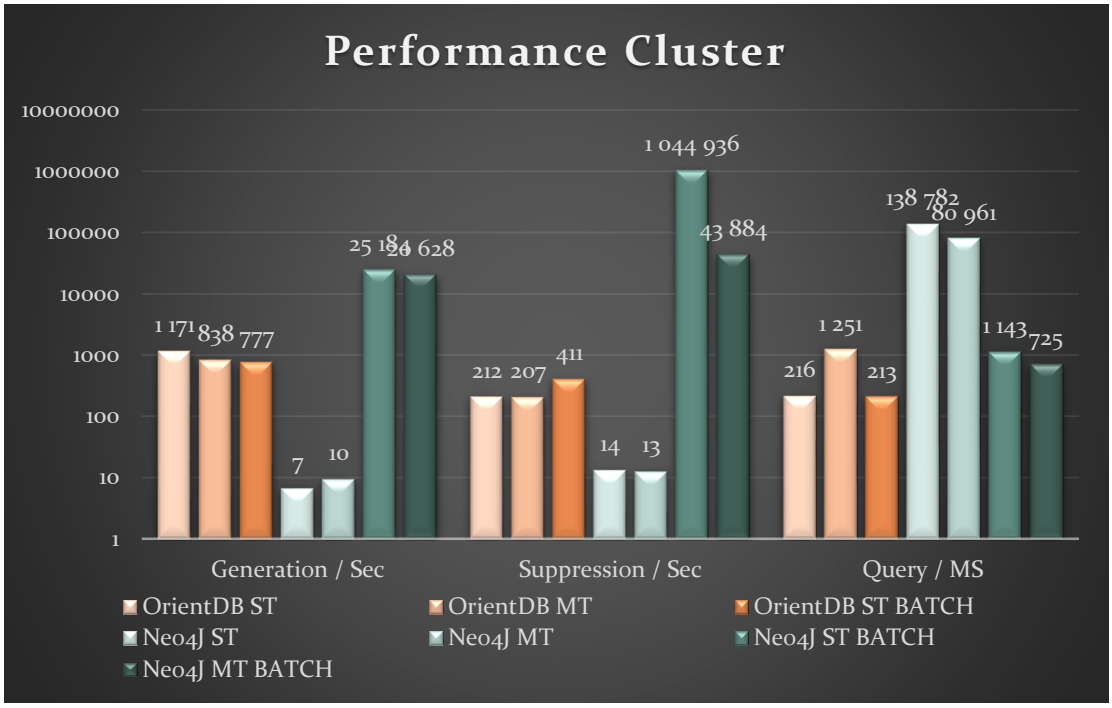

Image 24 – Performance en moyenne Architecture Distribuée

On peut voir que *Neo4J* est en tête mais seulement dans le cas où les transactions sont réalisées par paquet de nœuds et non pas nœud par nœud.

Il faut aussi souligner que l'architecture distribuée n'apporte pas une augmentation des performances en écriture mais plutôt en lecture. Même si cela reste discutable car la configuration matérielle de la machine sur laquelle les tests en single instance n'est pas la même que les machines du cluster.

CONCLUSION

Ce qu'il faut retenir de cette étude à propos d'*OrientDB* :

- Utilisation se fait par la Multi Model API développé par *OrientDB* qui permet d'être polyvalent sur les objets manier, de faire des transactions, et du semi-SQL, les autres API comme *Tinkerpop* sont considérées comme obsolète
- Implémentation de l'intégration de transactions en Multi-Thread conséquente complexe car intégrité des transactions se fait côté client et un simple changement de version de la base de donnée peut amener au refus de la transaction
- Des transactions tel que la suppression en Multi Thread de paquets de nœuds peut provoquer le verrouillage de certaines ressources et compromettre l'utilisation de la base de donnée
- Les performances sont uniformes, indépendamment de la taille de la transaction effectuée

- Architecture distribuée Haute disponibilité : Master – Master simple à mettre en place, possibilité de faire du *sharding* (assigner un cluster à un objet)
- Sous License Apache2 à part l'interface Web de Cluster management

Et de *Neo4J* :

- Utilisation se fait par l'API JAVA Driver qui permet de faire des transactions mais aussi soumettre des requêtes *Cypher*, un langage semi-SQL conçu spécialement pour *Neo4J*
- Il y a une gestion des transactions plus autonomes, les transactions ne sont pas rejetés si la base change en cours de route, seul un réel conflit peut amener au refus d'une transaction
- Intégrité des données mis en avant à travers la gestion précédemment des transactions.
- Performances dépendent de l'implémentation des transactions, il faut éviter de soumettre des transactions trop petites et privilégiées le regroupement des actions.
- Architecture distribuée Causal : Master - Slave est plus complexe à mettre en place qu'une architecture Haute Disponibilité et le fait que toutes les transactions passent par le Leader amène forcément une dysmétrie au niveau de l'utilisation des machines.
- Implémentation de l'architecture distribuée payante

Références

Neo4J : https://neo4j.com/

OrientDB : https://orientdb.com/

JanusGraph : http://janusgraph.org/

Bibliothéque Apoc Neo4J : https://neo4j-contrib.github.io/neo4j-apoc-procedures/#_job_management_and_periodic_execution

Image 1 : https://db-engines.com/en/ranking/graph+dbms

TinkerPop : http://tinkerpop.apache.org/

Cypher : https://neo4j.com/developer/cypher-query-language/

Cassandra : http://cassandra.apache.org/

FDS : https://logiciels.cnes.fr/en/cat%C3%A9gorie-de-logiciel/dynamique-du-vol