Προχωρημένα Θέματα Βάσεων Δεδομένων

Εξαμηνιαία Εργασία

Κάζδαγλη Αριάδνη - 03118838
Παράνομος Ιωάννης - 03118021
Ομάδα 28
Ακ. Έτος: 2023-24, 9ο Εξάμηνο

Github Project Page

Ζητούμενο 1

Η εγκατάσταση, πραγματοποιήθηκε σε τοπικό μηχάνημα με τους προτεινόμενους πόρους.

Οι web εφαρμογές έγιναν διαθέσιμες στο τοπικό δίκτυο και προσβάσιμες από τους παρακάτω συνδέσμους:

- HDFS: http://192.168.2.14:9870
- YARN: http://192.168.2.14:8088/cluster
- Spark History: http://192.168.2.14:18080/

Τα παραπάνω resources καταστήθηκαν προσβάσιμα στα μέλη της ομάδας για απομακρυσμένη σύνδεση μέσω Port Forwarding.

Ενδεικτικά παραθέτουμε απόσπασμα από το HDFS Web App, όπου φαίνεται ότι έχουμε πράγματι 2 ενεργούς κόμβους:

Summary

Security is off.

Safemode is off.

323 files and directories, 321 blocks (321 replicated blocks, 0 erasure coded block groups) = 644 total filesystem object(s).

Heap Memory used 92.45 MB of 128 MB Heap Memory. Max Heap Memory is 1.94 GB.

Non Heap Memory used 65.17 MB of 67.81 MB Commited Non Heap Memory. Max Non Heap Memory is <unbounded>.

Configured Capacity:	56.75 GB
Configured Remote Capacity:	0 B
DFS Used:	1.74 GB (3.06%)
Non DFS Used:	32.02 GB
DFS Remaining:	20.06 GB (35.35%)
Block Pool Used:	1.74 GB (3.06%)
DataNodes usages% (Min/Median/Max/stdDev):	1.30% / 4.82% / 4.82% / 1.76%
Live Nodes	2 (Decommissioned: 0, In Maintenance: 0)
Dead Nodes	0 (Decommissioned: 0, In Maintenance: 0)
Decommissioning Nodes	0
Entering Maintenance Nodes	0
Total Datanode Volume Failures	0 (0 B)
Number of Under-Replicated Blocks	0
Number of Blocks Pending Deletion (including replicas)	0
Block Deletion Start Time	Tue Jan 16 12:01:12 +0200 2024
Last Checkpoint Time	Tue Jan 16 12:01:13 +0200 2024
Enabled Erasure Coding Policies	RS-6-3-1024k

Ζητούμενο 2

Για το παρόν ερώτημα και για τα επόμενα, κάναμε χρήση του PySpark, αποθηκεύοντας το ζητούμενο Dataframe σε μορφή CSV στο DFS και έπειτα φορτώνοντάς το στα επόμενα ερωτήματα. Δίνονται τα ζητούμενα αποτελέσματα (αριθμός γραμμών και τύπος κάθε στήλης):

Τύπος στηλών στο τελικό Dataframe:

Column Name	Type
DR_NO	int

Column Name	Type
Date Rptd	date
DATE OCC	date
TIME OCC	int
AREA	int
AREA NAME	string
Rpt Dist No	int
Part 1-2	int
Crm Cd	int
Crm Cd Desc	string
Mocodes	string
Vict Age	int
Vict Sex	string
Vict Descent	string
Premis Cd	int
Premis Desc	string
Weapon Used Cd	int
Weapon Desc	string
Status	string
Status Desc	string
Crm Cd 1	int
Crm Cd 2	int
Crm Cd 3	int
Crm Cd 4	int
LOCATION	string
Cross Street	string
LAT	double
LON	double
ZIPcode	string

Column Name	Type
Community	string
Estimated Median Income	string

Πλήθος γραμμών στο τελικό Dataframe:

⇒ 2973193

Ζητούμενο 3

Για την εκτέλεση του Query 1 χρησιμοποιήσαμε δύο διαφορετικά αρχεία, εκκινόντας ένα Spark Session, με 4 Spark Executors, στο κάθε ένα. Και τα δύο παρήγαγαν το ίδιο αποτέλεσμα, βέβαια σε διαφορετικούς χρόνους.

Αποτέλεσμα Query 1:

Year	Month	count	Rank
2010	1	15750	1
2010	3	15653	2
2010	4	15516	3
2011	1	18269	1
2011	7	18257	2
2011	8	18041	3
2012	1	22983	1
2012	8	22832	2
2012	5	22525	3
2013	8	9112	1
2013	7	8956	2
2013	1	8682	3
2014	7	12106	1
2014	8	11990	2

Year	Month	count	Rank
2014	12	11969	3
2015	8	13857	1
2015	10	13502	2
2015	9	13365	3
2016	8	14596	1
2016	7	14479	2
2016	12	14323	3
2017	10	33585	1
2017	7	33357	2
2017	8	33297	3
2018	11	15613	1
2018	10	15341	2
2018	12	15198	3
2019	7	19122	1
2019	8	18979	2
2019	3	18856	3
2020	1	6090	1
2020	2	5869	2
2020	5	5530	3
2021	7	34744	1
2021	8	33653	2
2021	10	33639	3
2022	7	16748	1
2022	8	16526	2
2022	5	15833	3
2023	7	37594	1
2023	8	37500	2
2023	10	37033	3

Χρόνοι Εκτέλεσης:

Μέθοδος	Χρόνος Εκτέλεσης (s)
Dataframe	0.1399979591369629
SQL API	0.23599624633789062

Συμπεράσματα:

Αναφορικά στην επίδοση των δύο μεθόδων, το Dataframe API εμφανίζει καλύτερο χρόνο εκτέλεσης σε σχέση με το SQL API. Αυτό υποστηρίζεται από τη βιβλιογραφία, καθώς για περίπλοκα queries αναφέρεραι ότι πιθανώς το Dataframe API να έχει καλύτερη επίδοση. Όμως είναι σημαντικό να σημειωθεί ότι κατά κύριο λόγος και οι δύο μέθοδοι εν τέλει δεν διαφέρουν σημαντικά και άρα στις περισσότερες περιπτώσεις θα έχουν κοντινούς χρόνους εκτέλεσης. Ενδεικτικά και τα δύο APIs κάνουν χρήση του Catalyst Optimizer, που πραγματοποιεί βελτιστοποιήσεις στον κώδικα που έχουμε συντάξει, καθώς και In-Memory Processing που είναι απαραίτητο για γρήγορες πράξεις σε μεγάλα δεδομένα. Εν τέλει αυτό που κρίνεται πιο σημανικό για την σύγκριση των παραπάνω, είναι η εκοικείωση του προγραμματιστή με τον κάθε τρόπο (Python/ SQL), και η κατά περίπτωση ανάγκη για τμηματοποίηση και έλεγχο ενδιάμεσων αποτελεσμάτων, όπου το Dataframe API, είναι σαφώς πιο κατάλληλο.

Σημειώνουμε ότι σε αυτές καθώς και στις παρακάτω υλοποιήσεις όπου ζητείται χρόνος εκτέλεσης, έχουμε πάρει μετρήσεις στο κρίσιμο τμήμα της εφαρμογής, αφήνοντας εκτός των μετρήσεων το φόρτωμα και την εκτύπωση στοιχείων, που θα είναι και κοινά μεταξύ των υλοποιήσεων

Ζητούμενο 4

Όπως και στο παραπάνω, για την εκτέλεση του Query 2 χρησιμοποιήσαμε δύο διαφορετικά αρχεία, εκκινόντας ένα Spark Session, με 4 Spark Executors, στο κάθε ένα. Και τα δύο παρήγαγαν το ίδιο αποτέλεσμα, σε διαφορετικούς χρόνους.

Αποτέλεσμα Query 2:

TimeOfDay	#
Night	240393
Evening	188522
Afternoon	149713
Morning	125500

Χρόνοι εκτέλεσης:

Μέθοδος	Χρόνος Εκτέλεσης (s)
Dataframe	0.12399816513061523
RDD	13.30742597579956

Συμπεράσματα:

Όπως γίνεται εμφανές το Dataframe API, είναι σαφώς ταχύτερο από το RDD API, κάτι αναμενόμενο για περίπλοκα queries.

Ειδικότερα, το Dataframe API, ενώ κάνει χρήση του RDD (Resilient Distributed Dataset), είναι ουσιαστικά μια μέθοδος υψηλότερου επιπέδου, η οποία πριν καταλήξει στο RDD, εκτελεί μια σειρά από βελτιστοποιήσεις, ώστε να διασφαλίσει καλούς χρόνους εκτέλεσης. Αντίθετα το RDD API, εκτελεί τον κώδικά που παρέχεται στη μορφή που είναι, χωρίς να προσπαθεί να τον βελτιώσει. Αν και ο προγραμματιστής μπορεί να κάνει από μόνος τους βελτιστοποιήσεις στον κώδικα, μεριμνώντας για την ορθή και βέλτιστη σειρά των πράξεων και τη σωστή επιλογή μεθόδων, εν τέλει το Dataframe API, προσφέρει παραπάνω αυτοματισμούς, καλύτερη δομή και μεγαλύτερη ευκολία χρήσης. Τέλος σημειώνεται ότι ακόμα και στην παραλληλοποίηση την οποία αποζητούμε με 4 Spark Executors, το Dataframe API, είναι η προτεινόμενη δομη, λόγω της αποδοτικότητας που εμφανίζει στη διαχείριση πόρων

Ζητούμενο 5

Η εκτέλεση του Query 3 πραγματοποιήθηκε με χρήση του Dataframe API.

Αποτέλεσμα Query 3:

Victim Descent	#
Hispanic/Latin/Mexican	1032
Black	983
White	652
Other	62

Χρόνοι εκτέλεσης

Executors	Χρόνοι Εκτέλεσης (s)
2	4.475822687149048
3	3.3158769607543945
4	3.1158902645111084

Συμπεράσματα:

Όπως γίνεται φανερό από τους χρόνους εκτέλεσης, η αύξηση των Spark Executors, προσφέρει καλύτερη επίδοση! Παρόλαυτά, το αποτέλεσμα αυτό δεν μπορεί να θεωρηθεί καθολικός κανόνας. Η αύξηση των Spark Executors, προσφέρει μεγαλύτερη παραλληλοποίηση στην εφαρμογή μας, όμως αυτό σημαίνει ότι πρέπει να εξετάσουμε και άλλες παραμέτρους όπως οι πόροι που θα χρειαστούν για αυτό και αν τους διαθέτουμε. Έπειτα πρέπει να σκεφτούμε ότι σε μερικές εφαρμογές οι παραπάνω Executors μπορεί να προκαλέσουν αυξημένο Data Shuffling που είναι ανεπιθύμητο. Απο την άλλη πλευρά παραπάνω Spark Executors είναι πιθανό να μας προσφέρουν παραπάνω Data Locality το οποίο είναι επιθυμιτό για την εφαρμογή μας, αν και στις δικές μας περιπτώσεις όλα φαίνεται να εκτελέστηκαν σε *LOCALITY_LEVEL: PROCESS_LOCAL*.

Εν τέλει ο προγραμματιστής θα πρέπει να κρίνει κάθε φορά τα παραπάνω και να αποφασίζει πιο είναι το ιδανικό πλήθος Spark Executors.

Ζητούμενο 6

Η εκτέλεση του Query 4 πραγματοποιήθηκε με χρήση του Dataframe API.

Αποτέλεσμα Query 4a:

year	average_distance (km)	#
2010	2.745	6762
2011	2.719	7797
2012	2.91	2.7458600
2013	2.686	2981
2014	2.732	3411
2015	2.626	4557
2016	2.681	5389
2017	2.719	13592
2018	2.677	5776
2019	2.74	7129
2020	2.412	2492
2021	2.663	18240
2022	2.473	7660
2023	2.668	17572

Αποτέλεσμα Query 4b:

Station_Division	average_distance (km)	#
77TH STREET	2.638	17383

Station_Division	average_distance (km)	#
SOUTHEAST	2.106	13691
NEWTON	2.028	9884
SOUTHWEST	2.698	8625
HOLLENBECK	2.652	6100
HARBOR	4.067	5798
RAMPART	1.578	4990
NORTHEAST	3.864	4320
OLYMPIC	1.823	4251
HOLLYWOOD	1.442	4158
MISSION	4.718	4003
FOOTHILL	3.774	3536
WILSHIRE	2.398	3519
NORTH HOLLYWOOD	2.678	3507
CENTRAL	1.136	3469
WEST VALLEY	3.44	3194
PACIFIC	3.751	2677
VAN NUYS	2.221	2670
DEVONSHIRE	3.981	2557
TOPANGA	3.446	2072
WEST LOS ANGELES	4.197	1554

Ζητούμενο 7

Πραγματοποιήσαμε τα ζητούμενα, ενώνοντας σε ένα αρχείο τα βήματα για τα Query 3 και Query 4, μέχρι και την υλοποίηση των απαραίτητων join. Έπειτα διαμορφώσαμε τις αντίστοιχες συναρτήσεις, ώστε αντί να παράγουν τα αποτελέσματα των join, να μας δίνουν 2 dataframes, κάθε φορά, ώστε να παράξουμε εμείς με ελεγχόμενο τρόπο τα

join. Δημιουργήσαμε 4 Spark Sessions, ένα για κάθε μέθοδο και τυπώσαμε για αυτά λεπτομέριες για τα join με χρήση `explain()` και επίσης παρατηρήσαμε τις διαφορές τους με χρήση του Spark History Server: Jobs - SQL/Dataframe - Details

Συνοπτικά, παρουσιάζουμε μια μικρή εξήγηση και χρήση για την κάθε στρατηγική.

Broadcast Join

Το broadcast join χρησιμοποιείται όταν ένα από τα δύο Dataframes, είναι αρκετά μικρό, ώστε το Spark να μπορεί να το κάνει broadcast σε όλους τους κόμβους-εργάτες. Αυτό έπειτα επιτρέπει να γίνουν οι απαραίτητες πράξεις τοπικά, και έτσι μειώνεται το data shuffling. Αυτή η μέθοδος προτιμάται στην περίπτωση που έχω ένα πραγματικά μικρό Dataframe, καθώς σε άλλες περιπτώσεις μπορεί να δημιουργηθεί σημαντικό overhead, αλλά και να μειωθεί η απόδοση της μνήμης (κατειλλειμένοι πόροι).

Σημειώνεται ότι σε κάθε περίπτωση τα επιμέρους μέρη του Dataset μας, είναι επαρκώς μικρά ώστε να χωράνε στη μνήμη του κάθε executor.

Merge Join

Το Merge Join, αρχικά ταξινομεί το κάθε Dataframe, βάσει του key που θα συνδυαστεί. Έπειτα, στα δύο ταξινομήμένα Dataframes, επιχειρεί να βρεί κλειδιά που ταιράζουν και έπειτα τα ενώνει (merge), προχωρόντας στις επόμενες γραμμές. Η στρατηγική αυτή χρησιμοποιείται για μεγάλα Dataframes, καθώς δεν απαιτείται να φορτωθεί κάθε Dataframe ολόκληρο στη μνήμη ενώ λειτουργεί καλά και σε περιπτώσεις παραλληλοποίησης. Δεν προτείνεται η χρήση της σε περίπωση που έχω ένα σημαντικά μικρότερο Dataframe, καθώς ίσως να καθιστά την ταξινόμηση του μεγάλου, μη συμφέρουσα.

Shuffle Hash Join

Αρχικά τα δεδομένα από κάθε Dataframe, χωρίζονται βάσει του key τους και τα δεδομένα με το ίδιο key, τοποθετούνται στο ίδιο partition, στον ίδιο κόμβο. Έπειτα κατασκευάζεται ένα Hash Table, συνήθως για το μικρότερο Dataset, σε κάθε κόμβο, όπου αποθηκεύονται ζεύγη κλειδιού με τα αντίστοιχα δεδομένα του. Έπειτα γίνονται queries στο Hash Table, το οποίο επιστρέφει τα αντίστοιχα δεδομένα και έτσι αυτά συνδυάζονται.

Η στρατηγική αυτή χρησιμέυει για μεγάλους όγκους δεδομένων, καθώς δεν απαιτεί μεγάλα Dataframes να είναι φορτωμένα στη μνήμη.

Shuffle Replicate NL Join

Αντίστοιχα με πριν, δεδομένα με το ίδιο κλειδί καταλήγουν σε partitions, στον ίδιο κόμβο. Αντί όμως τώρα να κατασκευάσουμε Hash Table, το μικρότερο Dataframe, αντιγράφεται σε κάθε κόμβο και συγκρίνεται με το μέρος των δεδομένων του μεγαλύτερου Dataframe, πυο περιέχει ο εν λόγω κόμβος. Αυτό γίνεται με ένα Loop, όπου για κάθε γραμμή του μεγάλου Dataframe, εξετάζουμε όλες τις γραμμές του μικρότερου. Έτσι αυτό ζρησιμοποιείται στην περίπωτση που ένα Dataframe, είναι πολύ μεγάλο για να χωρέσει στην μνήμη ενός κόμβου, ενώ το άλλο είναι επαρκώς μικρό.

Συμπεράσματα:

Το Dataset, με το οποίο εμείς δουλέψαμε είναι επαρκώς μικρό και χωράει εύκολα στην μνήμη του κάθε Worker-Node. Ενδεικτικά το Dataset μας είναι **721M**, ενώ το κάθε μηχάνημά μας έχει **8G** RAM και έχουμε ορίσει **spark.driver.memory=1G**. Αυτά μας οδηγούν στη μέθοδο που και το ίδιο το Spark επέλεγε για κάθε προηγούμενο join, χωρίς να ορίσουμε κάτι με το `hint()`, δηλαδή το **Broadcast Join.**