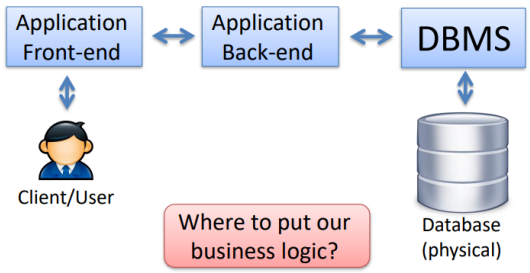
:The Four V's of Big Data;

1. Volume - נפח נתונים גדול.
2. Velocity - איכות הנתונים.
3. Variety - מגוון נתונים.
4. Veracity - אמינות נתונים.

חוק MOORE – כל שנתיים הכוח החישובי מכפיל את עצמו. היום השיפור פחות משמעותי בגלל החסם הפיזיקלי. (יש חסם עליון לגודל האפשרי). פתרון אחר לכך הוא עבודה בצורה מקבילית.

ארכיטקטורת 3 השכבות הסטנדרטית (שכבות לוגיות)



Frontend – הצד שעובר מול הלקוח, המשתמש, צד שאין בו הרבה לוגיקה.
Backend – הצד של הלוגיקה העסקית, אלגוריתמים.
DB – בו מאוחסנים הנתונים.

יתרונות המודל:
מודולריות – הפרדה בין השכבות, ניצור צוות לכל שכבה, נוכל לבצע שינויים בצורה בלתי תלויה בשכבות האחרות.

חסרונות:
כפל קוד - מספר אפליקציות הניגשות לאותו מידע, כל אפליקציה ניגשת באופן שונה ולכן תהיה קיימת כפילות קוד. דבר המהווה בעיה בעת ביצוע שינויים (נצטרך לבצע שינויים בכל האפליקציות).
אבטחה – אבטחת הטבלה אליה ניגשים (ממספר מקורות שונים) תיפגע.

פתרון לבעיות המודל – Stored Procedures

Persistent procedures/functions are stored locally and executed by the database server.
נגדיר פונקציות ופעולות שונות בתוך בסיס הנתונים ונחסוך קריאות חיצוניות ייחודיות של כל אפליקציה.
הבדל בין פונקציה לפרוצדורה: פרוצדורה לא חייבת להחזיר ערך.

יתרונות:

1. אין כפילות קוד (נגזר ששינויים יתבצעו פעם אחת, הקוד פשוט וקריא יותר).
2. קוד פשוט יותר + ניתן לקרוא לפונקציות ששמורות בDB באמצעות כל שפת תכנות (אין צורך בשינוי syntax).
3. חסכון בתקשורת (רק קריאה של פונקציות).
4. יותר מאובטח.

חסרון: לכל DB יש שפה משל עצמו, לכן ברגע שנרצה לעבור בין שפות של SQL (SQLite, MySQL וכו') תיווצר בעיה ונצטרך לבצע שינויים רבים.

תכנות DB

Embedded Commands – פקודות DB כתובות בתוך הקוד שנכתב ב-backend – בשפת התכנות (שאינה SQL) נשלב פקודות של SQL.
Library of DB functions – חבילות המאפשרות לתקשר עם בסיס הנתונים, API.

שלבים בתכנות DB:

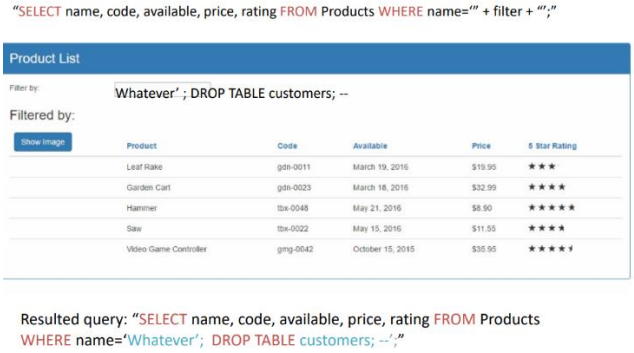
1. יצירת חיבור מול בסיס הנתונים.
2. כתיבת שאילתות וקבלת תוצאות.
3. סגירת החיבור עם בסיס הנתונים.

Embedded SQL:

נכתוב את פקודות ה-SQL בשפת התכנות בה נעבוד (java, python, c).
נשתמש בפקודות של execute ופקודות של פתיחת וסגירת חיבורים (connection) של השפה כאשר את השאילתה עצמה נכתוב כמחרוזת שתתקבל כקלט לפעולת ה-execute.

Dynamic SQL:

הגדרה של שאילתה באופן דינאמי – לא ידוע מראש איזה פרמטרים השאילתה תקבל או לחילופין מה היא תבצע.
מתן אפשרות למשתמש להכניס ערכים נדרשים בעצמו. כלומר להשתמש בפרמטרים בשאילתה שיוכנסו לאחר מכן ע"י המשתמש.
דרך פעולה זו יוצרת בעיות אבטחה – SQL injection:



התממשקות בין java לבין בסיס הנתונים באמצעות JDBC (ללא ORM).

PLSQL

שפה פרוצדורלית, כמו JAVA. משלבת SQL כדי לשלוט ולעבד את המידע. מאפשרת יותר פונקציונליות מאשר SQL. מאפשר לעבד את המידע על שרת ה-DB כדי לחסוך תעבורה וזמן בין שרת ללקוח.

מורכבת ממשתנים וקבועים (constants), פרוצדורות, פונקציות, טריגרים, SP וטיפול ב-exceptions (catch).
בנויה ב-block structure: חלק הצהרתי, חלק ביצועי וחלק המטפל ב-exceptions. רק החלק הביצועי נחוץ בהכרח.

כללים, Syntax ופעולות:

- צריך להכריז על משתנים לפני שמשתמשים בהם.
- השמה מבצעים ע"י := ולא רק =.
- מחרוזות ומערכים מתחילים באינדקס 1 ולא 0.
- אם משתנה הוגדר כקבוע אז אסור לשנות אותו.
- מחרוזות מסומנות בגרש אחד בכל צד ולא גרשיים (שניים).
- אופרטור || משמש לחיבור מחרוזות (concatenation).

- NVL – אם הערך הראשון של הארגומנט הראשון הוא NULL, תשתמש בארגומנט השני.
- לא CASE SENSITIVE.
- כל IF נגמר ב-END IF. גם לולאות (LOOP).
- אין הזחות.
- DECODE – כמו IF ELSE (לכן גם יכול לטפל ב-NULL).

Procedure ב-PLSQL: לא חייבת להחזיר ערך. כל פרמטר מוגדר ע"י שם, read/write property וסוג (טיפוס).

<u>Read/write property:</u>

IN – חייב להיות ערך למשתנה בזמן הקריאה לפרוצדורה (ברירת מחדל, המשתנה הינו read only).
OUT – הפרוצדורה עצמה חייבת להזין ערך למשתנה (מאותחל ב-NULL, write only).
IN OUT – לפרמטר יהיה ערך לפני הקריאה והפרוצדורה בתורה תיתן לו ערך חדש (read & write).

Function ב-PLSQL: חייבת להחזיר ערך + לציין איזה טיפוס מוחזר.

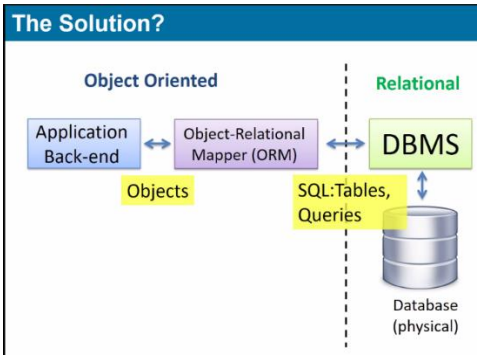
Trigger ב-PLSQL: פרוצדורה המוגדרת before/after פעולות insert/update/delete.

<u>שימוש ב-old ו-new:</u>

1. **INSERT**- :old.value= NULL, :new value= post insert value
2. **DELETE**- :old.value= Pre Delete value, :new value= null
3. **UPDATE**- :old.value= Pre update value, :new value= Post Update value

ORM – Object Relational Mapping

יש קושי בין שפות של תכנות מונחה עצמים (תומכים בהורשה, בקשרים מורכבים וכו') לבין גישה לDB רלציוני, לכן קיימים מתווכים שמקשרים בין הדברים (אובייקטים לטבלאות).
ORM – רעיון שמיושם ע"י כלים שונים שנועדו לתת מענה לאותה ההמרה. מתווך בין הטבלאות לבין האובייקטים.



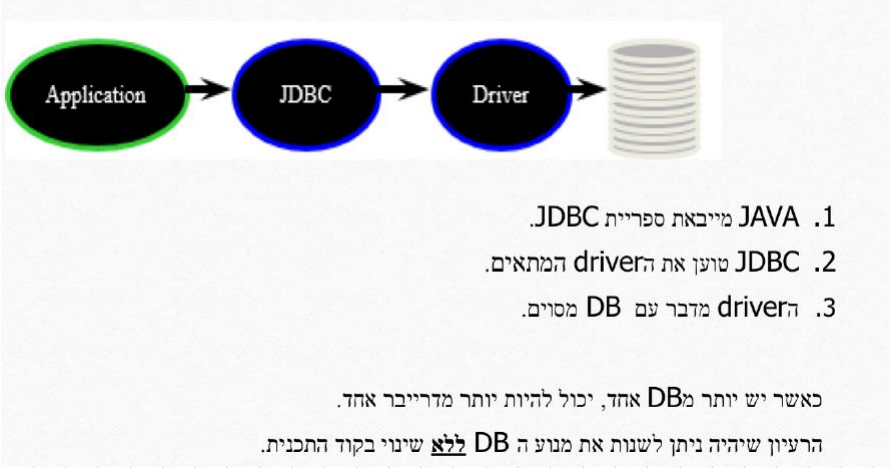
JDBC

דוגמה לספריית קוד שלא עובדת בשיטת ORM. זהו ממשק שכל DB מממש בעצמו. מגדיר אובייקט מסוג result set.

<u>דוגמה לשיטת עבודה:</u>

1. הגדרת ה-driver.
2. הגדרת הכתובת (הנתיב למיקום) שבה נמצא ה-DB.
3. טעינת ה-driver שאיתו עובדים ופתיחת חיבור באמצעות פקודת getconnection.
4. אם יש הרשאות והחיבור הצליח, מקבלים אובייקט connection, שדרכו מייצרים אובייקט מסוג statement. אז אפשר להגדיר את השאילתה (מחרוזת הנשמרת למשתנה).
5. (exectuequery(sql – מקבל את השאילתה ומחזיר אובייקט מסוג result set – אפשר לעבור עליו באופן איטרטיבי.
6. סגירת החיבור.

driver – רכיב תוכנה המחבר בין אפליקציות/אפליקציה לרכיב חומרה.
JDBC – JAVA API.
JDBC driver – בעזרתו יוצרים connection ומגדיר פרוטוקול עבודה בין ה-client ל-DB.



Connection:
טעינת ה-driver – Class.forName, יצירת instance של driver וחיבור ל-DriverManager.
יצירת החיבור – DriverManager.getConnection – ארגומנטים: driver, connection URL, שם משתמש וסיסמה.

Statment: יצירת גישה ל-DB, בדר"כ ע"י שאילתות SQL (כמחרוזת – statement לא מקבלת פרמטרים).

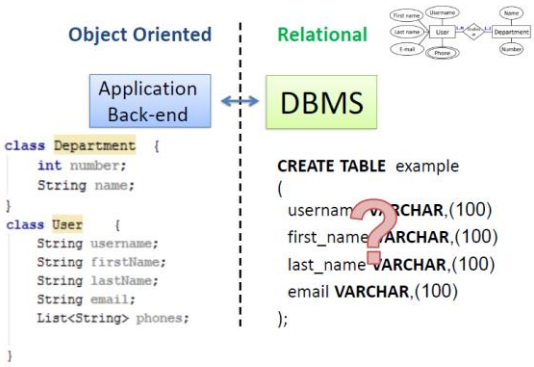
PreparedStatement: משתמשים כאשר רוצים להשתמש בשאילתות SQL **באופן חוזר**. מקבלת פרמטרים בזמן ריצה. ? – מסמל קבלת פרמטר.

CallableStatement: משתמשים כדי לבצע stored procedures. מקבלת פרמטרים בזמן ריצה.

Execute Statement: executeQuery (select), executeUpadte (update, insert, delete).

Result Set: תוצאה המוחזרת משאילתות כ"רשימת" תוצאות בשורות עם cursor לשורה הנוכחית (מתחיל בהצבעה לשורה הראשונה). ניתן לגשת לעמודות בשורה ע"י שם/מספר עמודה. כאשר סוגרים statement אז ה-result set נמחק.

Impedance Mismatch – האופן שבו המודל של ה-object oriented עובד שונה מהאופן שבו הdatabase הרלציוני עובד (בגלל קשרים והורשות).



יש כלים יותר מתקדמים מ-JDBC שיכולים לנצל את היכולות של ה-object oriented ולהפחית את המאמץ.
ORM בא לפתור את בעיית חוסר ההתאמה בין ה-java objects (object oriented) לבין טבלאות ה-RDBMS.
זה רעיון שניתן לממש ע"י כל מיני כלים.

Hibernate

ספריית ORM שפותחה עבור java.
Table ---> Class, Row ---> Object, Column ---> Properties.
Hibernate משתמש ב-XML שמבצע מיפוי (קובץ קונפיגורציה).
session – החיבור הפיזי שפותחים עם ה-DB (Session Pool – syntax).
Query – השאילתה בעצמה.
Transaction – רצף של פעולות שמבצעים בשלמות/לא מבצעים בכלל (לדוגמה העברה בנקאית).
ניתן לשלב גם טריגרים.

היינו רוצים לוותר על הצורך ב-XML (כי זה עוד קובץ שצריך לנהל) ולהגדיר את המיפוי בקוד עצמו – ניתן לעשות זאת Java Annotations (Hibernate Annotations). כמו הערות כאלה שבאמצעותם רואים את המיפוי.
יתרונות – יותר מובן כי רואים את המיפוי מול העיניים ואין צורך בקובץ נוסף.

מיפוי קשרים:
דוגמה – אחד לרבים: @onetomany (@ - סימן של annotation).
הגדרת סוג fetch:
Lazy – אם ביקשנו אובייקט שיש בתוכו connection (חיבור לאובייקטים נוספים בגלל קשר אחד לרבים) אז הוא לא יאותחל כל עוד לא ביקשנו get על ה-collection. נקבל את השדות המורכבים רק בדרישה. יותר יעיל וקל.
Eager – כשמבקשים אובייקט כל השדות שלו יוחזרו מראש ויאותחלו מראש, נקבל מראש גם בלי לבקש.

Hibernate Query Language (HQL) – שפת שאילתות מונחית עצמים. פחות שימושית אבל מאפשרת לכתוב שאילתות מורכבות כמו ב-SQL עם כל מיני קיצורים.

Criteria – מאפשר ביצוע כל מיני שאילתות עם Restrictions. בנוסף גם ביצוע שאילתות של or/and עם אובייקט LogicalExpression, הוספת השאילתה לקריטריון וקבלת התוצאות מהקריטריון לתוך רשימה.

Java API for ORM – JPA.

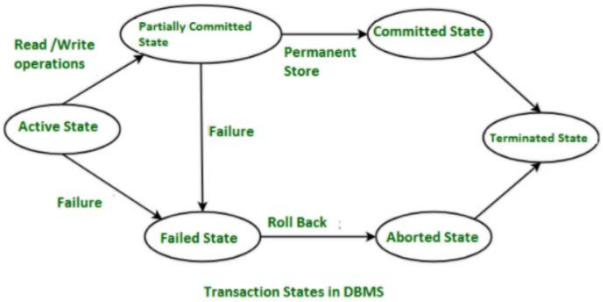
טרנזקציות

טרנזקציה – אוסף פעולות שמטרתן לבצע פעולת על אחת, יחידה לוגית אחת. טרנזקציה מתבצעת בשלמותה או לא מתבצעת כלל.

Single-User System: בכל רגע נתון קיים משתמש יחיד במערכת.
Multiuser System: מאפשרת למספר משתמשים לבצע פעולות במקביל.
Concurrency: מקביליות.
Interleaved processing: מערכת בעלת מעבד יחיד. אם יש מספר תהליכים במקביל אז כל תהליך מבצע

את פעולתו בחלקים ובכל שלב חלק יחיד של תהליך יחיד מבצע את הפעולה על גבי המעבד.
Parallel processing: מערכת בעלת מספר מעבדים. מעבד מוקצה לתהליך וכך הדברים מתבצעים בצורה מקבילית.

מחזור החיים של טרנזקציה:



כשטרנזקציה נכנסת למערכת היא נכנסת למצב פעיל – Active State.
1. אם היא מצליחה היא עוברת למצב של partially committed state (בדרך ביצעה פעולות קריאה וכתיבה).
השינויים שהטרנזקציה עשתה נשמרים באופן מקומי ב-DB server ולאחר מכן היא עוברת למצב של committed state – השינויים שעשתה נשמרים בדיסק.
* partially committed – הטרנזקציה טרם התבצעה בשלמותה ולכן אם התרחש כשל, נרצה לבטל את כל פעולות הטרנזקציה שהתבצעו עד כה.
2. הטרנזקציה כשלה –עוברת למצב של failed state, היא לא הושלמה ולא ביצעה commit ולכן נרצה לבטל אותה (אם היא הספיקה לעשות שינויים כלשהו נרצה לבטלם, כי היא לא נעשתה במלואה) ולכן היא תעבור למצב Aborted State באמצעות פעולת Roll Back.

תוכנית:
- קוראת נתונים ממסד הנתונים.
- מבצעת חישובים בזיכרון הפנימי המוקצה לה.
- כותבת נתונים חדשים למסד הנתונים.

כל תוכנית נכתבת תחת ההנחה שהיא מסיימת את פעולתה בצורה ותקינה ושהתוכנית פועלת לבדה (בלתי תלויה בתוכניות אחרות).

ביצוע התוכנית:
ביצוע סדרתי – הטרנזקציות רצות אחת אחרי השנייה, בכל פעם מתבצעת תוכנית אחת בלבד בשלמותה.
ביצוע מקבילי – התוכניות משולבות זו בזו, בכל נקודת זמן מתבצעת פעולה בודדת של אחת התוכניות.

יתרונות הביצוע המקבילי:

1. נמנע מעיכובים הנובעים מהמתנה לתוכניות אחרות.
2. ניצול מקסימלי של משאבי המחשוב.

חסרונות הביצוע המקבילי:

1. התוכניות עצמן לא יודעות שהן מתבצעות באופן מקבילי לתוכניות אחרות.
2. שגיאות העלולות להיווצר מהביצוע המקבילי של התוכניות.

התאוששות
Database recovery is the process of restoring the database to a correct (consistent) state in the event of a failure.

נרצה שבמקרה של כשל במערכת לשמור על נכונות הטרנזקציות. טרנזקציה שעשתה commit, תשתקף ב-DB (השינויים שביצעה) וטרנזקציה שלא עשתה commit, לא תשתקף. כך נשמור על עקביות המערכת.

השינויים מתבצעים באופן לוקאלי: נשים לב שהשינויים אותם הטרנזקציות מבצעות לרוב מתבצעים ב- main memory ורק בנקודות זמן מסוימות הנתונים נשמרים בדיסק, לכן נצטרך לשים לב לכך בביצוע ההתאוששות.
יתרון: כך נוכל לחסוך במשאבים ובזמן, העתקה של הנתונים לדיסק היא פעולה שצורכת משאבים רבים ולכן ככל שנפחית את כמות ההעתקות לדיסק, כך נקצר את זמן הריצה.
(חיסרון) אם נכתוב כל פעולה של הטרנזקציה לדיסק והטרנזקציה תקרוס באמצע, נצטרך למחוק את כל הכתיבות שביצענו לדיסק – בזבוז משאבים.

מטרת ההתאוששות: נרצה לשמר את עקביות המערכת לאחר הקריסה.

למה התאוששות נחוצה?
טרנזקציות יכולות להתבטל ממגוון סיבות:
ע"י המשתמש – החליט לבטל את הטרנזקציה/הכניס קלט לא תקין, ע"י ה-DBMS – ביטול טרנזקציה, למשל כשמגיעים למצב של dead locks, בגלל טעות חישובית – למשל חלוקה ב-0.
המערכת יכולה לקרוס: תקלה ב-DBMS, כשלים של מערכת ההפעלה/רכיבי חומרה/דיסק.

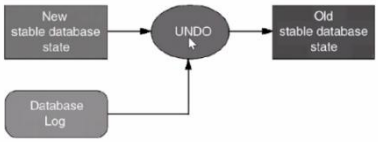
עקרונות ACID: לפיהם פועלים מסדי נתונים רלציוניים.
Atomicity – טרנזקציה מתבצעת במלואה או לא מתבצעת כלל.
Consistency – השינויים משתקפים תמיד במערכת.
Isolation – כל טרנזקציה מבוצעת באופן נפרד מהאחרות.
Durability – אם משתמש מקבל חיווי שהטרנזקציה בוצעה היא אכן חייבת להתבצע.
העקרונות אותם נרצה לשמר באמצעות מנגנון ההתאוששות הם: Atomicity ו-Durability (Consistency ו-Isolation מטופלים ע"י מנגנון strict 2 phase locking).

איך זה מתבצע? Commit, rollback.

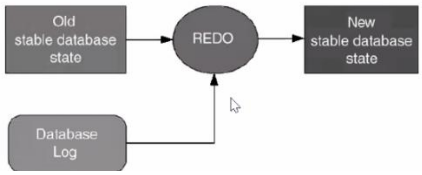
Log-based Recovery
Write ahead logging. Write first to log, and then to disk.
לפני כל פעולה שנכתבת לדיסק, ראשית נכתוב אותה ב-log. למעשה ה-log משמש כתיעוד לפעולות בלבד.

<u>הנחות:</u>
כתיבה של בלוק לדיסק היא אטומית.
כתיבה של בלוק יחיד מתבצעת בזמן סביר.

ביצוע UNDO: עבור הפעולות שלא ביצעו commit – נרצה לבטלן, כלומר להחזיר את השינויים שביצעו למצבם המקורי.



ביצוע REDO: עבור הפעולות שביצעו commit אבל טרם נכתבו לדיסק – נרצה לכתוב אותן לדיסק.



LOG

LSN	Type	Tid	PrevLSN	Page	BFIM	AFIM
1	UP	1	0	A	0	1
2	UP	1	1	B	0	2
3	**UP**	**1**	**2**	**C**	**0**	**N**

מבנה ה-log:

LSN – מספר סידורי של הפעולה.
Type – סוג הפעולה.
Tid – מספר סידורי של טרנזקציה.
PrevLSN – הLSN של הפעולה הקודמת של הטרנזקציה Tid.
Page – הדף בזיכרון אותו נעדכן.
BFIM – ערך ה-page לפני העדכון.
AFIM – ערך ה-page לאחר העדכון.

הודעת commit בלוג קובעת אם טרנזקציה הסתיימה:
אם כתוב commit, אז עושים REDO במקרה של תקלה. אם לא כתוב, אז עושים UNDO.

מתי מעבירים את המידע שנשמר באופן לוקלי ב-DBMS לדיסק? נשאף לאיזון בין כמות הגישות לדיסק לבין כמות הנתונים הנשמרים בזיכרון הלוקלי בזמן נתון ("סתימת הזיכרון").

Force: בכל commit נבצע כתיבה לדיסק, כלומר המערכת צריכה לעצור ולבצע העתקה לדיסק לאחר ביצוע commit של כל טרנזקציה. פעולה זו היא יקרה ומאטה את המערכת.
No Force: המערכת לא בהכרח תכתוב לדיסק לאחר כל commit שמתבצע.

Steal: מאפשר לכתוב לדיסק גם אם הטרנזקציה לא ביצעה commit. על מנת לאפשר steal נצטרך להיות מסוגלים לבצע UNDO.
No Steal: לעולם לא נבצע עדכון בדיסק עבור טרנזקציות שלא ביצעו commit, כך נמנע מצורך בביצוע UNDO (חסרון – עומס בזיכרון הלוקלי).

	Steal	No Steal
Force		Commit= Flush
No Force	Best performance	

הביצועים הטובים ביותר: **Steal – No Force** אך מקשה על התאוששות.

* הפעולה **Flush** משמשת להעברת המידע מהזיכרון הלוקלי לדיסק.

אלגוריתם ARIES

קלט – log לאחר קריסת מערכת.
פלט – DB תקין.

מבנה:

Log – הבנוי כמתואר מעלה.
מבחינת הזיכרון הראשי:
Transaction Table – העמודות שלה הן Tid, PrevLSN ו-Status. טבלה זו מתארת עבור כל טרנזקציה מה הרשומה האחרונה שתועדה עבורה ב-log (PrevLSN) ואת סטטוס הטרנזקציה (הסתיימה commit, פעילה active).
Dirty Pages Table – העמודות שלה הן Page ו-LSN. מתארת את המשתנים שעודכנו בזיכרון הראשי אך עדיין לא נכתבו לדיסק.
Data (Table) (טבלה אחת לזיכרון הראשי ואחת לדיסק) – העמודות שלה הן Page, Img, LSN. מכילה לכל משתנה מה הערך שלו ומה ה-LSN הרלוונטיים עבורו ב-log.

Flush – פעולה המעתיקה את טבלת ה-Data השייכת לזיכרון הראשי אל הדיסק ומוחקת את תכולת טבלת ה-Dirty Pages (מימוש עקרון Steal).
**** ייתכן שיתבצע FLUSH רק לדפים מסוימים ולא לכולם.***

Check Points: המטרה – הימנעות מפעולת REDO לכל ה-Log.

1. נחכה שכל הטרנזקציות הפעילות יעשו commit ונעצור את הטרנזקציות הבאות מלהתחיל את פעולתן.
2. נעתיק את טבלאות ה-dirty pages וה-transactions לדיסק (נשים לב שכל המידע שעבר לדיסק הוא מידע מעודכן לאחר סיום פעולת הטרנזקציה).
3. נשמור CP (Check Point) ב-Log.

יתרון: בביצוע שחזור לא נצטרך לחזור לתחילת ה-Log אלא רק ל-CP האחרון.
חסרון: ביצוע CP מאטה את פעילות המערכת ועוצרת אותה בזמן העדכון.
כלומר ביצוע ה-Check point באופן זה אינו פיזיבלי, כי לא ניתן לעצור את כל המערכת לטובת עדכון זה.

The Complete ARIES Algorithm:

הנחות:
1. Write Ahead Log - צריך לוודא כתיבת פעולה ל-log לפני שהפעולה נכתבת לדיסק.
2. Data pages מאוחסנים בדיסק עם ה-LSN הרלוונטי.

האלגוריתם המלא כולל Fuzzy Check Points ומאפשר לקיים טרנזקציות חדשות בזמן שאנחנו מבצעים את תהליך ה-check point.
Fuzzy Check Points:

1. עם תחילת התהליך נכניס רשומה לLog שתיקרא Begin CP, נניח שנקודת זמן זו נקראת T ולוקח לה להתבצע X זמן.
2. בX זמן של ביצוע הCP קורות פעולות נוספות במערכת. לכן, כל מה שקרה לפני זמן T ישתקף במערכת בצורה טובה, אך הפעולות שהתבצעו בטווח [T,X+T] לא משתקפות בדיסק.
פעולות אלו ימשיכו להתקיים בזמן ביצוע ה CP באופן רגיל, כלומר הטבלאות מתמלאות כרגיל והפעולות מתבצעות.
3. כשנסיים את כתיבת הCP, נוסיף רשומה לLog שתיקרא End CP. נצרף את טבלת הDirty pages וטבלת הטרנזקציות לסוף הLog (כי במקרה של קריסה הזיכרון הראשי מתרוקן).
4. **בעת קריסת המערכת, נבצע שחזור החל מהנקודה Begin CP האחרונה הקיימת בLog.**

שלב ה-Analysis: מטרה - שחזור טבלת הDirty pages וטבלת הטרנזקציות (בדיקה אילו טרנזקציות סיימו את פעולתן ואילו לא).

1. תחילה "נטען" את טבלת ה-Dirty Pages וטבלת ה-Transactions מסוף ה-log (אם נשמרו כאלה ע"י ה-check point האחרון שבוצע בשלמותו).
2. נשמור במשתנה N את ה-LSN המינימלי מטבלת ה-Dirty Pages שטענו מסוף ה-log (לטובת שלב ה-REDO בהמשך).
3. נכניס ל-undo_set את הטרנזקציות שלא ביצעו commit.
4. **נתחיל את האנליזה החל מ-Begin CP של ה-checkpoint האחרון שבוצע בשלמותו**, ונמשיך עד לסוף הLog.
5. נעבור על כל הרשומות ב-log בזו אחר זו ונבצע אותן – נעדכן את טבלת הDirty pages ואת טבלת הטרנזקציות באופן הבא:
   a. אם "פוגשים" טרנזקציה חדשה שלא קיימת ב-undo_set – נוסיף אותה.
   b. אם טרנזקציה ביצעה commit, נמחק אותה מה-undo_set.
6. בסיום שלב זה טבלת הטרנזקציות תשוחזר באופן מדויק. טרנזקציות שנשארו במצב Active בסוף המעבר על הLog יופיעו ב-undo_set.

נשים לב שטבלת הDirty pages **לא** תשוחזר באופן מדויק (כיוון שייתכנו ביצועי flush שלא מגולמים בLog, אך פער זה אינו מפריע לביצוע ההתאוששות).

שלב ה-Redo: מטרה - שחזור טרנזקציות שבוצעו בהצלחה (ביצעו commit) אך לא נשמרו בדיסק.

1. נתחיל מ-$LSN = N$ ב-Log.
2. נבדוק האם הרשומה צריכה להיכתב לדיסק או לא, לפי הכללים הבאים:
   a. אם
      i. הרשומה הנוכחית ב-log מתייחסת ל-**Page שלא נמצא ב-Dirty Pages Table** (כנראה הדף נכתב כבר לדיסק ע"י flush)
      <u>או</u> (ה-page נמצא בטבלת ה-dirty pages)
      ii. אם $\boldsymbol{LSN(Page)^{DP} > LSN(Page)^{log}}$ (כנראה שהתבצע עדכון מאוחר יותר לדף),
      אז נדלג על רשומה זו ולא נבצע כתיבה לדיסק.
      <u>אחרת,</u>
   b. נייבא את נתוני ה-Page מהדיסק. אם $\boldsymbol{LSN(Page)^{Disk} < LSN(Page)^{log}}$**, אז נבצע REDO!** (כי הדף לא קיים/מעודכן בדיסק).
   c. אם 2 התנאים (a, b) לא מתקיימים – **לא נבצע כתיבה לדיסק** (השינויים של הרשומה על ה-page כבר משוקפים).
3. **במידה וביצענו REDO –** נמחק את הPage הרלוונטי מטבלת הDirty pages.
4. בסיום שלב זה טבלת הData זהה לטבלה מלפני הקריסה.

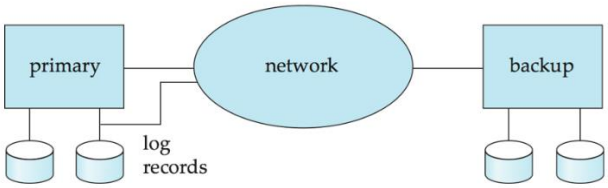
שלב ה-UNDO: מטרה - ביטול פעולות של טרנזקציות שלא ביצעו commit.
פלט משלב האנליזה: undo_set המכיל את הטרנזקציות שלא ביצעו commit (והן במצב Active).

<u>עבור כל טרנזקציה השייכת לundo_set:</u>

1. נתחיל מהLSN הגדול ביותר בLog ששייך לטרנזקציה הנמצאת בundo_set.
2. נבדוק מה היה ערך הpage לפני ביצוע פעולה זו ונעדכן את הערך לערך זה (בטבלת הData).
3. נתעד את הביטול בLog באמצעות רשומה שתיקרא UNDO.
4. נעבור לPrevLSN המתאים ונחזור לשלב 2.
5. נמחק את כל הטרנזקציות שהתבטלו מטבלת הטרנזקציות.

<u>גיבוי</u>

תקלה עוצמתית יותר מאשר מצב שדורש התאוששות – גיבוי נדרש עבור מצב בו הקריסה מתרחשת בדיסק, כלומר המידע שבדיסק אבד וצריך לשחזר אותו, או לחילופין קריסה משמעותית וצורך בסיפוק שירות ללקוח גם בזמן הקריסה.



ארכיטקטורה לפתרון הבעיה:
Primary Site – האתר בו נמצאים המידע והפעולות שמתבצעות, שולח עדכונים שנעשו לBackup Site ומתעד פעולות בlog.
Backup Site – בעת קריסה בPrimary Site ניתן לעשות שחזור לנתונים באמצעות הbackup (כולל שחזור באמצעות ה-log וע"י redo וundo). כך למעשה המערכת משתמשת בגיבוי כדי לחזור למצב בו הייתה לפני הקריסה.

יתרונות השיטה:

1) גם במקרה של קריסה הגיבוי מאפשר זמינות – higher availability.
2) המידע מגובה באתר המשני במידה והמידע מהאתר הראשי נאבד.

משימות הBackup Site:

1) Failure Detection – זיהוי נקודת כשל (של הPrimary Site).
זיהוי זה מתבצע באמצעות:
   a. heartbeat messages – אותות חיים שנשלחים לbackup מהprimary, כך הbackup מעודכן במצבו של הprimary כל הזמן.
   b. תחזוקת מספר ערוצי תקשורת בין הprimary וה-backup (למקרה שאחד הערוצים יקרוס).
2) Transfer of Control – העברת השליטה – אם הראשי קרס, להשתמש בגיבוי.
במידה וה-primary קרס, הגיבוי צריך לדעת "לתפעל" את תהליך השחזור, לטעון את הlog וה-DB האחרונים שנשלחו אליו, לבצע את הפעולות הנדרשות מהlog ולוודא את שלמות הטרנזקציות (באמצעות redo וundo).

Takeover Delay: פרק הזמן מרגע הקריסה ועד לרגע בו ה-backup כבר התחיל לעבוד ולטפל בקריסה.
איך נקצר את זמן זה?

1) Periodic Recovery – יצירת check points רבים בprimary site ושליחת עותקים שלהם לbackup.
דורש תקשורת רחבה מספיק ביניהם.
2) Hot Spare –
   a. נגדיר שאחרי כל מספר פעולות נבצע העתקה לגיבוי. באופן זה במקרה של קריסה לא קיימים תהליכים רבים לתקן וה-backup ישקף את מה שנמצא בprimary.
   b. ביצוע מידי של פעולות הlog בגיבוי – מאפשר את חוסר איבוד המידע, ברגע ביצוע commit על פעולה היא משתקפת בגיבוי.

אלטרנטיבה ל-Distributed Replication – Remote Backup:

Remote Backup	Distributed Replication
שרת ראשי + שרת גיבוי.	המידע מחולק למספר שרתים והפעולות מבוצעות בין השרתים.
אתר גיבוי שמתעדכן באופן תקופתי לפי תדירות לבחירתנו: גבוהה – התאוששות מהירה ומידע אמין אך צריכת משאבים גבוהה. נמוכה – התאוששות קשה, מידע לא עדכני אך חסכונית במשאבים.	כל רשומה נשמרת במספר עותקים. חסרון – דורש מקום אחסון, פחות רלוונטי היום. חסרון – עקביות consistency, בביצוע עדכון לרשומה יש צורך בגישה למספר שרתים.

ביצוע Commit של טרנזקציה

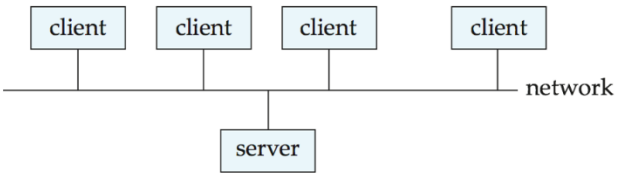
One-Safe: ברגע תיעוד פעולת הcommit בlog הנמצא בprimary, הטרנזקציה יכולה לבצע commit.
חסרון – בעת קריסת הprimary, במידה וה-primary לא שלח את הרשומה מהlog לbackup, לא קיים תיעוד של הcommit על אף שהמשתמש קיבל חיווי על commit וייתכן כי הפעולה תבוטל (ע"י undo). כלומר, לא נשמרת העקביות.
יתרון – מהירות המערכת.

Two-Very-Safe: ברגע שהטרנזקציה צריכה לבצע commit, עוד לפני שהמשתמש מקבל חיווי, הcommit מתועד בlog גם של הprimary וגם של הbackup ורק לאחר התיעוד יתבצע commit וישלח חיווי.
חסרון – **מתווספות נקודות כשל.** נצטרך לקבל חיווי מ2 מקומות שונים, אז אם קיים כשל תקשורת בין הbackup וה-primary הטרנזקציה לא תוכל לבצע commit.
יתרון – לא ייתכן מצב של חוסר עקביות.

Two-Safe: אם **התקשורת בין** הprimary וה-backup טובה, נפעל בשיטת Two-Very-Safe. אם **התקשורת ביניהם** לא טובה (קריסה בתקשורת), נפעל בשיטת One-Safe.

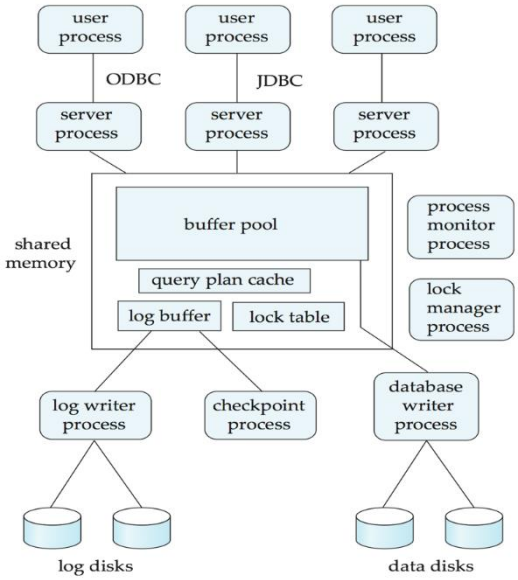
ארכיטקטורות Databases

ארכיטקטורה בסיסית – Client/Server: לכל לקוח יש זיכרון, דיסק, CPU עצמאיים. הלקוח מתקשר עם השרת דרך הרשת.



DBMS – מערכת שמערבת בתוכה DB:
frontend – ממשק משתמש רזה --> interface (SQL API) – דרכו מתקשרים עם הDB --> backend – מנוע SQL.

ארכיטקטורה ריכוזית: שרת יחיד המטפל במספר תהליכים.



- תהליך נפתח ע"י המשתמש, פותח תהליך הנקרא server process המתחבר לshared memory של השרת.

- בshared memory מתנהלים התהליכים הפתוחים. רכיבים:

1) buffer pool: משתנים גלובליים שימושיים, זיכרון זמני.
2) query plan cache: תפקידו לבנות מחדש שאילתות על מנת שיתבצעו בצורה אופטימלית.
3) lock table: בהנחה שיש מספר תהליכים המתבצעים במקביל ומבצעים שינויים במערכת, ננהל אותם באמצעות נעילות.
4) log buffer: ניהול כתיבה לlog ואת כתיבת הcheck points.

- **database writer process**: מנהל כתיבה לדיסק של השרת.

- **process monitor process**: תהליך שמטרתו לבצע בקרה על התהליכים הפועלים.

- **lock manager process:** תהליך שאחראי על ניהול הנעילות בDB כדי להקטין סיכוי לdeadlock.

<u>ארכיטקטורה מבוזרת:</u> מערכת מבוזרת היא מערכת בה המידע מבוזר על פני שרתים שונים או data centers שונים. לשרתים נקרא sites/nodes.

נעבור מארכיטקטורה ריכוזית למבוזרת – עדיף להשקיע בהרבה שרתים פחות חזקים ומתוחכמים לעומת השקעה בשרת ריכוזי מאוד חזק. כלומר נשתמש בחומרה לא חזקה בשילוב עם תשתית תקשורת מאוד טובה.

Cluster Computing: מספר גדול של commodity servers המחוברים ע"י commodity network.
<u>Rack</u> – מחזיק מספר קטן של שרתים (ארון שרתים).
<u>Data Center</u> – מכיל הרבה Racks.
דרך התקשורת היא יקרה ומהירה.
היררכיה מבחינת יעילות – בתוך שרת יחיד הפעולות הכי מהירות, בתוך rack התקשורת פחות מהירה וב-data center הכי איטית.

מערכת מבוזרת יוצרת עותק של אותו פריט מידע מספר פעמים. אם למשל יהיו 3 עותקים נעדיף לשמור 2 עותקים בתוך אותו rack את השלישי בrack אחר, כך גם נאפשר תגובה מהירה וגם גיבוי בrack נוסף (במקרה וה-rack ש-2 העותקים נמצאים בו נשרף למשל).

<u>סוגי רשתות</u>

Local Area Network – LAN: רשת מקומית, החומרה נמצאת במקום יחיד ובדרך כלל מחוברת פיזית. יש יציאה אחת מהרשת "החוצה" האחראית לנהל את התקשורת מחוץ לLAN. מתאים לרשת קטנה, מצומצמת גיאוגרפית.
Wide Area Network – WAN: תקשורת בין אתרים שונים – התבסוסות על אינטרנט/לווין. התקשורת בין האתרים מערבת ערוצי תקשורת שונים עם אמינות משתנה וקצב שידור שונה. רשת כזו מאפשרת לנהל סניפים/data centres של אותה חברה שנמצאים במקומות שונים בצורה יחסית טובה.

<u>יתרונות ואתגרים של שרתים המתנהלים בצורה מבוזרת</u>

<u>מדדי ביצוע:</u>
<u>Response Time</u>: זמן ביצוע משימה מרגע הגשת הבקשה ועד לקבלת חיווי ע"י המערכת.
<u>Throughput</u>: כמות המשימות שניתן להשלים בקטע זמן נתון.

<u>Scale Up VS Speed Up:</u>

Speed Up – ייעול זמן ביצוע של פעולה מסוימת. כלומר מדידת ושיפור מהירות הביצוע על בעיה שנשארת בגודל קבוע – אותה פעולה בפחות זמן. משפר את הResponse time.
ההנחה היא שאם נגדיל את מספר המעבדים, כך יגדל באופן לינארי גם מספר הביצועים. בפועל לרוב הגידול הינו sub linear (כלומר פחות טוב).

$SpeedUp = \frac{runtime\ in\ small\ system}{runtime\ in\ N-times\ larger\ system}$ - אם שווה N – אז ה-speedup לינארי.

<u>מצב לינארי מניח הנחות שלא מתקיימות ולכן הגידול אינו לינארי:</u>

1. המידע לא מחולק באופן אחיד על פני השרתים.
2. חוק הזנב הארוך – עומס על מספר מועט של פרטי מידע.
3. ניהול שרתים דורש משאבים וזמן.

Scale Up – יכולת לטפל בכמות בעיות גדולה יותר בפרק זמן נתון, כלומר זמן הביצוע קבוע וכמות המשימות בזמן זה משתנה. משפר את הThroughput.
בהגדלת כמות הDB והשרתים, היינו מצפים שכמות הטרנזקציות המטופלות פר שניה יישאר זהה. בפועל, כמות הביצועים קטנה בקצב sublinear.

$ScaleUp = \frac{runtime\ of\ small\ system\ on\ small\ problem}{runtime\ of\ N-times\ larger\ system\ on\ N-times\ larger\ problem}$ – אם שווה 1, ה-scaleup לינארי.

Batch VS Transaction Scale Up:

Batch Scale up – היכולת לבצע משימה אחת גדולה באופן מהיר יותר. דורש מחשב חזק.

Transaction Scale Up – היכולת לעבד מספר טרנזקציות קטנות בפרק זמן נתון. דורש כמות משאבים גדולה.

הסיבות לפער בין הרצוי למצוי:

Startup Cost: התנעת תהליכים רבים וניהולם הוא מסובך יותר ודורש זמן רב יותר = יקר.

Interference: תחרות על משאבים משותפים שצריך לנהל (נעילות). בנוסף התקשורת מסובכת יותר במערכות גדולות.

Skew: קיים שוני בזמני הביצוע של התהליכים השונים, כלומר המערכת רגישה לזמן הסיום של "החולייה החלשה בשרשרת".

שני סוגי Scaling עיקריים:

Vertical Scaling – חיזוק חומרה בשרת מסוים. טוב לבעיות הנפתרות במחשב יחיד (***שלא ניתנות למקבול***).

חסרון - קיים גבול לכוח שניתן לתת ולהעמיס על שרת בודד.

Horizontal Scaling – הוספת שרתים נוספים, נטל העיבוד מתחלק בין שרתים.

חסרון – מורכבות מימוש במערכת מבוזרת.

ארכיטקטורה של DB מבוזרים

Shared Memory: שרתים נפרדים כאשר לכל שרת CPU ודיסק משלו, החולקים זיכרון RAM משותף. בזיכרון המשותף נשמר מידע שיש צורך שיהיה משותף לכל השרתים (למשל משתנים גלובליים).

חסרונות – קשה לבצע Scaling, יש מגבלה לכמות הזיכרון הנשמרת לRAM ולכן הגדלת מספר הבקשות מוגבל.

יתרון – הגישה לRAM היא מהירה ולכן נוכל לזרז תהליכים.

Shared Disk: שרתים נפרדים, לכל שרת CPU משלו, RAM משלו אבל הדיסק הקשיח משותף. כלומר כל השרתים חולקים את אותו Data בדיסק. נשתמש באינדקס גלובלי המכוון את השרתים למידע הנדרש מהדיסק.

חסרונות:

1. קריסה בDisk מקריסה את כל השרתים.
2. תקשורת רבה, גישה לדיסק היא יקרה.

Shared Nothing: לכל שרת יש מעבד, זיכרון RAM ודיסק קשיח משלו. הם מחוברים באמצעות תקשורת ביניהם. ארכיטקטורה זו נפוצה בdata centers. המשימות מתחלקות בין השרתים, אם שרת צריך מידע שיושב בשרת אחר, הם מתקשרים ביניהם ומעבירים מידע.

חסרונות – ***ניהול מורכב, שמירה על עקביות***, שמירה על תקשורת חזקה.

יתרונות:

1. ארכיטקטורה המאפשרת Scaling טוב.
2. מערכת שרידה, אם שרת נופל האחרים יכולים לפצות עליו.
3. קל לחזק את המערכת ולהוסיף לה שרתים.

Hierarchical: משלבת את הארכיטקטורות האחרות.

חסרון – קושי בניהול מערכות הבנויות בארכיטקטורות שונות.

סוגי DB מבוזרים

הומוגניים: בכל השרתים בDB קיימת אותה תוכנה, כלומר אותה מערכת הפעלה ואותו DBMS. הגישה לDB נעשית באמצעות ממשק המדמה שימוש בDB ריכוזי. המטרה היא ליצור מערכת מבוזרת המתנהלת כDB אחד.

יתרונות:

1. חלוקת עומסים בלי להעריך את היכולות של כל שרת בנפרד.
2. אין צורך לבצע התאמות בין השרתים.
3. נוחות למשתמש (ע"י אבסטרקציה – כאילו DB אחד).

הטרוגניים: לאתרים שונים בהם נשמר הDB יש מערכות הפעלה שונות, חומרה שונה וDBMS שונה.
המטרה היא לחבר בין הDB כדי לתת פונקציונליות שימושית למערכת.
חסרון – החיבור והתקשורת ביניהם מורכבים ומסובכים.

Tradeoff במערכות מבוזרות

Shared Data: הtradeoff העיקרי הוא האופן בו השרתים חולקים מידע. העברת המידע קובעת את יעילות המערכת.
טרנזקציה לוקלית – ניגשת לdata שנמצא באותו השרת/data center/rack (פחות נפוץ).
טרנזקציה גלובלית – יש צורך בשיתוף פעולה בין שרתים הנמצאים בdata centers שונים.

Autonomy: כל data center מנהל באופן עצמאי את עצמו, כלומר הוא מתחייב לספק את המידע שקיים אצלו בזמן סביר.

Redundancy: כל רשומה נשמרת במספר שרתים שונים ובדרך כלל בdata centers שונים.

יתרונות:

1. Availability – ניתן לבצע חלוקה של העומס בין שרתים המחזיקים את אותו מידע שהם מעוניינים לשלוף (במקום ששרת יחיד יצטרך לשרת את כל הדרישה בעצמו).
2. שמירת המידע בdata centers נוספים עוזר לשרידות. במקרה של קריסה המערכת עדיין יכולה לתפקד.

חסרון: מורכבות – צריך לנהל את כל המערכת הזו הכוללת חלוקת עומסים, טיפול בנפילות וכו'.

יתרונות וחסרונות DB מבוזרים

יתרונות:

1. ביצועים טובים יותר.
2. זמינות גבוהה יותר.

חסרונות:

1. עקביות המידע – תחזוקת הרשומה מתפרשת על data centers שונים לכן כשנבצע עדכון הוא יהיה צריך להיות מבוצע במספר מקומות שונים.
2. שימור atomicity וdurability קשה ליישום.

ביצוע שאילתות בצורה מקבילית בDB רלציוני

שיטות:

1. Partition Parallelism – מספר שרתים כך שכל שרת עובד על חלק נתונים אחר. ניתן לחלק את המידע בין השרתים לפי סדר מסוים.
2. Connection Parallelism – כל שאילתה מתבצעת בצורה סדרתית אך נאפשר למספר שאילתות להתבצע במקביל. ניתן בצורה זו להגדיל הספק.

3. Pipeline Parallelism – שאילתה המורכבת ממספר תהליכים, נבצע כל תהליך במקביל. שיטה מורכבת כיוון שיש צורך באי תלות בין התהליכים.

חלוקת הנתונים בין השרתים השונים

סוגי שאילתות:

1. **Full table scan** – מעבר על כל הטבלה, זו האופרציה הכבדה ביותר ונשתדל להימנע ממנה.
2. **Point queries** – שאילתות ממוקדות, איתור רשומות ספציפיות.
3. **Range queries** – נחזיר רשומות בטווח מסוים, יש צורך בסריקת הטבלה בהתאם לחלוקת המידע בין השרתים.

שיטות חלוקת נתונים:

1. **Round Robin Partitioning:** כל רשומה מתחלקת בצורה אקראית לשרתים זמינים. scaling בעייתי.
יתרון – המידע מתחלק בצורה אחידה בין השרתים מבחינת כמות הרשומות, מאפשר ביצוע תהליכים מקביליים.
התאמה לסוגי שאילתות –
- לא מתאימה לRange, כיוון שתתבצע בצורה לא יעילה, כיוון שאין סדר בין השרתים.
- מתאימה מאוד לFull scan כיוון שנוכל להשתמש בשרתים במקביל לסריקה.
2. **Hash Partitioning:** שימוש בפונקציית Hash הממפה רשומה לשרת מסוים לפי תוצאת פונקציית hash. ניתן לבצע scaling טוב ע"י עדכון פונקציית hash.
התאמה לסוגי שאילתות –
- מתאימה מאוד לpoint כיוון שנוכל לחפש לפי hash את הרשומה באופן כמעט ישיר.
- יעילה לFull אם השדה עליו מבצעים hash מגוון יחסית כך שהמידע מפוזר בין השרתים באופן אחיד יחסית.
- לא מתאימה לrange כי החלוקה אינה לפי טווחים.
3. **Range Partitioning:** כל שרת מאחסן רשומות לפי טווח מסוים, לפי השדה אותו נגדיר.
חסרונות:
- נשים לב שהחלוקה יכולה להיות לא מאוזנת.
- עשוי להגביל את היכולת למקבל פעולות על הנתונים.
התאמה לסוגי שאילתות -
- לא מתאימה לfull כי החלוקה יכולה להיות מאוד לא אחידה, ייתכן מצב ששרת אחד מסיים הרבה לפני שרת אחר.
- מתאימה לrange אם השאילתה של הrange זהה לחלוקה הנתונה, כך נדע איפה כל טווח נמצא.
- עדיפה מבחינת זמן ריצה לעומת HASH כאשר רוצים לשלוף טווח ערכים קטן (מעט בלוקים) מתוך טבלה.

- **Round robin** partitioning
  - + Best balancing
  - – Scaling is a problem
  - – Range queries not supported
- **Hash** partitioning
  - + Scales well, supports elasticity
  - + Efficient balancing
  - – Range queries not supported
- **Range** partitioning
  - – Can be imbalanced
  - + Range queries are built in
  - – For large ranges, can limit I/O parallelism

הערות:

- בתכנון נכון נוכל להימנע מגישות מיותרות לשרתים בעת ביצוע השאילתות.
- מידע עדכני נעדיף לשמור באופן מבוזר ומידע היסטורי נעדיף לשמור בצורה ריכוזית.
- בOracle: קיימת חלוקה לפי hash, לפי range ולפי list partitioning – לכל חלוקה מוגדרים ערכים בדידים ספציפיים לפיהם נחלק.

אינדקסים: יתרון – שליפה מהירה. חסרון – זמן יצירה, מקום אחסון.

1. אינדקס לוקלי – אינדקס שניתן להגדיר עבור חלוקה ספציפית ולא עבור טבלה מלאה (למשל רק עבור תקופת הזמן האחרונה, כי ידוע שזה מידע שנשלף באופן תדיר).
2. אינדקס גלובלי – אינדקס על המידע בכל השרתים.

חלוקה מאוזנת:

חלוקה לא מאוזנת נקראת Skewed Distribution – כאשר יש חלוקה לא אחידה של המידע, נוטה להיות מוטה/לא מאוזנת (למשל כשיש יותר שמות משפחה באותיות כ' ול' (כהן ולוי מאוד נפוצים) ואז המידע לא אחיד).

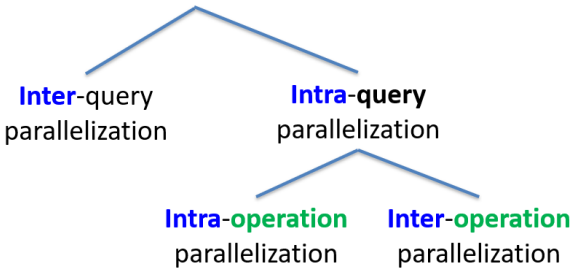
סיבות לחלוקה לא מאוזנת:

1. חלוקה לא מספיק טובה – בחירה לא טובה של פרמטרים.
2. שימוש בפונקציית Hash על שדה לא מתאים.
3. המידע עצמו לא מתחלק בצורה שוויונית.

פתרונות:

1. Balanced range partition – מיון ויצירת היסטוגרמה למידע על בסיס השדה המחולק.
2. Virtual processor partitioning – יצירת מעבדים "וירטואליים", חלוקת המידע לפי טווח למעבדים הווירטואליים, מיפוי מאוזן של מעבדים ווירטואליים למעבדים הפיזיים.
   כל שרת מחזיק יותר מחלק לוגי אחד, למרות שפיזית זו חלוקה אחת (שרת אחד).

מקבול שאילתות



איך נמקבל שאילתות?

Inter-Query Parallelization

מקבול בין שאילתות שונות, כלומר מספר שאילתות ירוצו במקביל, **נותן מענה לscale up ול-throughput**.

ביצוע inter query בארכיטקטורות שונות:

1. Shared Memory – ארכיטקטורה נוחה למימוש של Inter query.
   יתרון – כשמספר שאילתות מתעסקות עם אותו data, נשמור אותו בshared memory ונאפשר ריצה מקבילית של השאילתות עם גישה לזיכרון זה.
   חסרון – גודל הmain memory מוגבל. כלומר ארכיטקטורה זו הטובה ביותר אך אינה פיזיבלית.
2. Shared Disk – אם יש הרבה שאילתות הניגשות לטבלה מסוימת הדיסק מתקשה לשרת את כל השאילתות, הוא צוואר בקבוק במערכת ויכול לקרוס מהעומס.
3. Shared Nothing – חוסר איזון. שאילתות ארוכות ירוצו לאט וקצרות ירוצו מהר. בעיה נוספת היא שמירה על עקביות.

לסיכום, המימוש של inter query יותר קל ב-shared memory, אבל זו ארכיטקטורה לא פיזיבלית ולכן עובדים בshared disk או ב-shared nothing, אך יש לכך חסרונות, כפי שציינו.

Cache Coherency: עקביות של הCache – עקביות המידע. נרצה שעדכון פריט יתבצע וישתקף גם בשרתים האחרים.

1. Shared Disk – לפני ביצוע של כתיבה/קריאה ננעל את המשתנה, נקרא את ערך המשתנה מהדיסק המשותף, נבצע שינוי בשרת, נכתוב את העדכון לדיסק ורק לבסוף נשחרר את המנעול. חסרון – תהליך איטי, פוגע בזמינות, נדרשים לגשת לדיסק הרבה ומונע מקביליות של שאילתות על אותו פריט.
   זהו פתרון פשטני שמתאים יותר למערכת שלא מטפלת בהרבה שאילתות.
2. Shared Nothing – בארכיטקטורה זו כל שרת הוא ישות עצמאית ומבצע שאילתות באופן לוקלי. כל פריט למעשה שמור במספר מקומות ולכן קיים קושי בשמירה על העקביות בין השרתים השונים, נצטרך לוודא שמירה על העקביות בכל עותקי הפריט.

Intra-Query Parallelization

מקבול שאילתה ספציפית, שיפור ביצועי השאילתה הספציפית. **נותן מענה לspeed up ול- response time.**

1. **Inter operation** – מקבול האופרציות השונות בתוך אותה השאילתה. למשל ביצוע של order by במקביל לselect. הפעולות צריכות להיות בלתי תלויות.
2. **Intra operation** – נמקבל כל פעולה (עצמה) בתוך השאילתה, כלומר את הפעולה עצמה ולא מספר פעולות במקביל (ביצוע הפעולות הוא סדרתי). למשל בביצוע where נעבוד לפי טווחים, כך שכל שרת מקבל טווח ובו הוא מבצע סינון. ע"י הוספת מעבדים – קל יותר לביצוע.

מקבול פעולות רלציוניות: הנחות – Shared Nothing, שאילתות של קריאה בלבד, n מעבדים וn דיסקים.
פעולות אפשריות – **Intra-Operation Parallelization:**

1. Sort – Range partition, Parallel External Sort-Merge.
2. Join.
3. Selection: סינון (where).
4. Projection: בחירה (Select).
5. Aggregation – פעולות אגרגציה.

ביצוע שאילתות בצורה מקבילית Intra-Operation

SORT

1. **Range Partition:**
   a. בהתחלה, הטבלה מחולקת באופן לא ממוין בין השרתים.
   b. כל שרת מקבל טווח ערכים.
   c. כל שרת סורק את הערכים הקיימים אצלו ושולח את הערכים לשרת המתאים להם (לפי הטווחים שהוגדרו).
   d. לאחר החלוקה המחודשת כל שרת מבצע מיון לטווח הערכים שקיבל.
   e. לאחר המיון יתבצע איחוד בין השרתים השונים.
2. **Sort Merge:**
   a. בהתחלה הטבלה מחולקת באופן לא ממוין בין השרתים.
   b. כל שרת ממיין באופן מקומי את המידע הקיים אצלו.
   c. נגדיר טווח וכל שרת ימפה את הרשומות שאצלו לפי הטווח הנוכחי לשרת המתאים שהוגדר לטווח.
   d. ההעברה תתבצע בעזרת batch (לוקחים אוסף של רשומות ומעבירים אותן בבת אחת).
   e. כל שרת ימיין את הערכים שהגיעו אליו לאחר החלוקה לטווחים.
   f. לבסוף נבצע מיזוג לפי סדר השרתים.

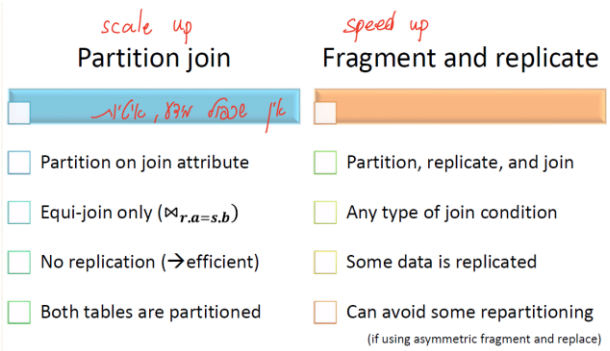
ההבדל העיקרי בין השיטות הוא המיון המקומי המתבצע בsort merge לפני החלוקה לטווחים.

JOIN

1. **Partition Join:**
   בהינתן השאילתה וה-join הבא: $r \bowtie_{r.a=s.b} s$. *תמיכה בjoin של שוויון בלבד.
   על כל רשומה בR נפעיל Hash על שדה a, בנוסף על כל רשומה בS נפעיל Hash על שדה b. נמפה את הרשומות למעבד המתאים לפי הHash (במקום hash ניתן להשתמש בrange המוגדר למיפוי).
   כל מעבד יעשה join לפי הטווח השמור אצלו באופן מקומי, לא ייתכן מצב שרשומות מתאימות $r.a = s.b$ לא יהיו באותו מעבד, כך שלא שנפספס הצלבות של מידע לפי אופן פיזור הנתונים.
2. **Asymmetric fragment and replicate:** * תמיכה בכל סוגי הjoin.
   בהינתן טבלה S הקטנה משמעותית/נכנסת בשלמותה לשרת יחיד ובהינתן טבלה R שלא נכנסת בשלמותה.
   נשלח את S לכל אחד מהמעבדים, R תחולק למעבדים בחלקים. כל מעבד יבצע Join מקומי בין כל S לבין החלק של R שאצלו. נשים לב שהאיזון בין המעבדים קריטי, כדי שלא יקרה מצב בו מעבד אחד מסיים הרבה לפני האחרים.
3. **Symmetric fragment and replicate:**
   גם R וגם S גדולות מכדי להיכנס בשלמותן לשרת יחיד.
   נחלק את R לn חלקים ואת S לm חלקים, נזדקק לm*n שרתים לביצוע הjoin.
   כל חלק מR ישלח n פעמים לn שרתים וכל חלק מS ישלח m פעמים לm שרתים, כל שרת מקבל שילוב ייחודי של חלק מR וחלק מS ומבצע join מקומי על קומבינציה זו. נשים לב שכל חלק מS מצטלב עם כל חלק מR.
   מדובר בjoin יקר יחסית (חילוק טבלאות) ודורש משאבים (m*n מעבדים). ***שיטה מהירה ביותר ובזבנית ביותר.***

<u>הבדלים בין Partition Join ל-Fragment and Replace:</u>

- ב-F&R כל רשומה מושוות לכל רשומה מהטבלה ה-2 ולכן אפשר לתמוך בjoin עם תנאים שונים (לא רק שוויון).
- לפעמים ב-F&R אפשר להימנע ממצב של חלוקה, נמנעים מלחלק את S (שזה דבר יקר, ב-asymmetric).
- ב-partition בכל מקרה תמיד חייב לחלק את כל הטבלאות.



4. **Distributed Join:**

בהינתן Sשת בשרת אחד Rו בשרת אחר, נרצה לבצע join ללא שליחת הטבלאות בין השרתים.

- נשלח רק את השדה המשותף מR לS (רק עמודות רלוונטיות).
- נבצע Join בS לפי השדה המשותף (**Semi Join** – ***מקטין את כמות המידע המועבר ברשת ולא מצריך שכפול***).
- עבור השורות הרלוונטיות, נוסיף את שאר השדות של השורה מטבלה R.

5. **Join Using Bloom Filter:**

אם נרצה לעשות join על יותר מ2 טבלאות נעבור לשימוש בBloom filter.

תזכורת: k פונקציות Hash ומערך ביטים. בהינתן סט ערכים S, לכל ערך בסט נפעיל k פונקציות hash ונקבל עבור כל ערך את ערך הhash שלו, נדליק את הביט המתאים של כל ערך כזה במערך. בהינתן ערך חדש שמגיע j, כדי לבדוק האם הוא נמצא בסט נפעיל עבורו k פונקציות hash, אם כל k הערכים דולקים **כנראה** שj בסט (ייתכן False Positive), אם לא כל הערכים דולקים **בהכרח** j לא בסט (לא ייתכן False Negative).

יעיל מבחינה חישובית, חוסך בתעבורת נתונים בתשתית התקשורת אך עלול לייצר עבודה מיותרת.

אופן ביצוע הJoin:

נגדיר R_1 בS_1ו R_2בS_2. שרת S_1 בונה bloom filter על כל הרשומות בשדה לפיו עושים את הjoin.

שרת S_1 שולח לS_2 את מערך k שנוצר בbloom. שרת S_2 עובר על הרשומות אצלו ובודק אותן ביחס למערך k, את כל הרשומות התואמות למערך זה, נשלח לS_1. ייתכן שנשלח רשומות עודפות (מיותרות) בגלל הfalse positive.

כך חסכנו מעבר בין השרתים של רשומות מלאות ושלחנו רק מערך של bloom filter ובנוסף המהירות תשתפר כי חיפוש בbloom מהיר יותר מsemi join.

Selection

1. **אם התנאי הוא $a_i = v$:** נלך לשרת המכיל את הרשומות המתאימות (בהנחה שקיימת חלוקה לפי hash/range על שדה a_i) – פעולה בשרת אחד.
2. **אם התנאי הוא $l < a_i < v$:** בהנחה שקיימת חלוקה לפי hash/range על שדה a_i מספר שרתים יצטרכו לבצע את הפעולה באופן מקומי.
3. **כל מקרה אחר:** אם הטבלה לא מחולקת לפי השדה או שה-where מפוזר, כל השרתים יריצו את התהליך.

Projection

כל השרתים המחזיקים את הטבלה צריכים לפעול, אם השדה אינו ייחודי עלולות להיות כפילויות.

על מנת להוריד כפילויות:

1. כל שרת באופן לוקלי מוריד כפילויות והשרת המאחד מוצא כפילויות בין השרתים השונים.
2. כל שרת באופן לוקלי לא מתייחס לכפילויות אלא רק ממיין את הנתונים והשרת הראשי מסיר את הכפילויות (עבודה רבה לשרת הראשי).

נעדיף לעבוד בשרתים במקביל מאשר ששרת אחד יעבוד לבד.

Grouping/Aggregation

- למשל מציאת max של עמודה – כל שרת מוצא max מקומי ומעביר למרכזי. במרכזי נבחר max מבין כל המקסימליים.
- למשל מציאת ממוצע של עמודה – נמצא ממוצע של כל שרת, נשלח למרכזי ונבצע ממוצע משוקלל על כל השרתים.

אופטימיזציה של שאילתות

ישנן דרכים רבות להרצת שאילתה בצורה מקבילית, יותר מאשר הרצתה בצורה סדרתית.

יוריסטיקה ***(פתרון טוב בזמן סביר, כלל אצבע לקבלת פתרון טוב)*** **–**

1. לא להשתמש בinter אלא רק בintra – כלומר רק למקבל את הפעולות עצמן.
2. לבחור את התוכנית הסדרתית הכי יעילה ואותה למקבל (plan – מיוצר ע"י האופטימייזר, סדר פעולות לביצוע שאילתה).

עיצוב של מערכת מקבילית

1. טעינת מידע – איך טוענים באופן אופטימלי את המידע.
2. עמידות בפני כשלים – מנהל השאילתה מוודא את עמידות השרתים ומטפל בנפילות.
3. Redundancy – שכפול ועותקים של מידע בשביל עמידות.
4. בניית אינדקסים נכונים בצורה חיצונית (ו-offline), הטמעה למערכת עם התאמה לשינויים שהתרחשו במהלך הזמן.

בסיסי נתונים מבוזרים – שכפול וחלוקה

הבדל בין DB ריכוזי למבוזר:

- DB ריכוזי: יכול להכיל מספר שרתים אבל רק אחד השומר את המידע. שרת זה מהווה את נקודת הכשל של המערכת.
- DB מבוזר: מודל בו יש הרבה שרתים ולכל שרת דיסק משלו, כך שהמידע נשמר בצורה מחולקת בין הדיסקים. גם אם שרת אחד נופל המערכת עדיין מתפקדת.

Fragmentation

חלוקת המידע (למשל טבלה מסוימת) למספר שרתים.

יתרונות:

1. פיזיבלית זה לא אפשרי לשמור את כל המידע על שרת יחיד.
2. עיבוד מקבילי של השאילתות.
3. Smart positioning – מאפשר חלוקה חכמה של המידע (למשל לפי מיקום גיאוגרפי) על מנת לקצר את זמן השליפה.
4. Fault tolerance – יכולת המערכת להמשיך לתפקד במקרה של קריסה.

חסרונות:

1. *לא מספק גיבוי למערכת במקרה של קריסה.*
2. *לא מאפשר הפצה מהירה יותר של מידע רלוונטי לשרתים השונים שלוקחים חלק בעיבוד טרנזקציה (יש יותר שרתים בגלל החלוקה).*

קריטריונים לחלוקה נכונה:

1. Complete: לא אבד מידע בחלוקה (אובדן מידע הינו למשל פספוס של טווח בחלוקה לטווחים).
2. Reconstructable: על מנת שנוכל לאחד בין טבלאות, כלומר לבצע בנייה מחדש, חשוב לשמור על השדות הייחודיים של כל טבלה בכל חלוקה שלה.
3. Disjointness: אין כפילויות בטבלאות החלוקה.

סוגי חלוקה:

חלוקה אופקית – **Horizontal Fragmentation:**

1. Primary:
חלוקת מידע לפי טבלה מסוימת ולפי טווח שנגדיר, נפצל את הטבלה למספר טבלאות עם כל השדות כך שבכל טבלה רשומות שונות. עונה על כל הקריטריונים.
2. Derived:
נגדיר קריטריון חלוקה לפי טבלה אחרת. למשל בהינתן טבלת עובדים וטבלת שכר עובדים, נוכל לחלק את טבלת העובדים לפי רמת השכר שלהם כפי שמופיע בטבלה השנייה.
* נשים לב שהחלוקה לוקחת בחשבון את כל הערכים. כך נענה על כל הקריטריונים.

חלוקה אנכית – **Vertical Partitioning**: חלוקה של העמודות. נחלק את הטבלה לפי עמודות שנגדיר (***חלוקה נושאית***), כל חלוקה צריכה להכיל את המפתח של הטבלה המקורית כדי לאפשר שחזור. ***לא מקשה על locking*** *(כי זוהי חלוקה לא שכפול).*

חלוקה היברידית – Hybrid Fragmentation: משלבת חלוקה אנכית ואופקית. למשל, חלוקה של טבלת עסקאות לפי טווח שנים (אופקי) ולאחר מכן חלוקה לפי עמודות (אנכי).

Replication

שיכפול של המידע, למשל פריט מסוים משוכפל מספר פעמים ומוחזק במספר שרתים/מיקומים שונים.

למה לשכפל? יתרונות:

1. גיבוי במקרה קריסה – ניתן לגשת למידע אם יש קריסה של שרתים.
2. זמינות המערכת – ממשיכה לתפקד בעת קריסה.
3. מאפשר עיבוד מקבילי – המידע נמצא במספר שרתים ולכן לא נוצר עומס על שרת אחד.
4. חסכון בתעבורה ותקשורת – מכיוון שהמידע זמין במספר שרתים, נחסכת גישה להרבה שרתים.
5. מיקומים גיאוגרפים שונים – נשמור חלק מהעותקים במיקומים גיאוגרפים שונים.

איך נשכפל? 3 גישות עיקריות:

1. No Replication – ללא שכפול, כל פריט מידע נמצא בשרת אחד.
יתרון: אחסון, שמירה על עקביות (בעדכון צריך לעדכן פריט יחיד).
חסרון: לא רלוונטי למערכת מבוזרת.
2. Partial Replication – כל פריט מידע משוכפל לחלק מהשרתים.
יתרון: שומר על המידע במקרה קריסה. בשימוש כיום.
3. Full Replication – מתאים למערכות קטנות, כל פריט מידע משוכפל לכל השרתים.
יתרון: שומר על יתרונות החלוקה.
חסרון: מתאים למערכות קטנות כי זה דורש הרבה מקום אחסון.

הבטחות של Distributed RDBMS

1. שיפור זמינות ואמינות המערכת – Availability/Reliability.
2. שיפור ביצועים - Performance.
3. ניתן לבצע scaling – יותר בקשות בזמן נתון.
scale אנכי – הוספת חומרה לשרת.
scale אופקי – הוספת שרתים ויכולת תקשורת – יותר פרקטי עבור מערכות מבוזרות.
4. **ניהול שקוף – Transparent management (למרות שהמידע מבוזר, מחולק ומשוכפל).**

Transparency

שקיפות המידע – המשתמש לא רואה את ביזור / חלוקת / שכפול המידע. קיימות רמות שונות לשקיפות.
סוגי שקיפות:
Fragmentation Transparency – שקיפות החלוקה. על אף שהמידע מחולק על פני שרתים שונים, המשתמש לא ירגיש זאת – כאילו המידע שמור בשרת אחד.

Location Transparency – שקיפות המיקומים. המידע מחולק על פני מיקומים גיאוגרפים שונים, המשתמש לא ירגיש זאת – כאילו המידע שמור באתר הלוקלי של המשתמש.
Replication Transparency – שקיפות השכפול. המשתמש מרגיש כאילו הוא עובד עם עותק יחיד.
במידה והעקביות לא נשמרת העיקרון של הreplication transparency לא נשמר.

איך נשמור על הTransparency?

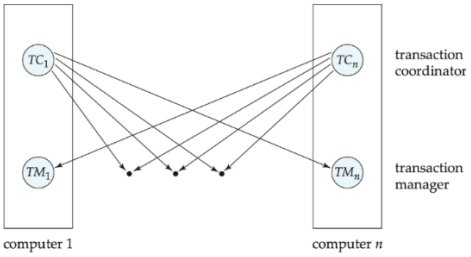
פתרון #1: **ייצור מפתחות ייחודיים –** לפי התנאים הבאים: לכל רשומה ישנו מפתח ייחודי, לכל שרת יש את האפשרות למצוא רשומות באופן לוקלי בצורה מהירה וליצור רשומות חדשות בצורה עצמאית. בנוסף, במקרה בו נרצה לשנות מיקום פריטים זה ייעשה באופן שקוף למשתמש.

1. הגישה הריכוזית – Centralized Name Server – אם כל שרת היה מייצר לעצמו מפתחות, ייתכן שהיו כפילויות ביצירת המפתחות ולכן נשתמש בשרת מרכזי האחראי על חלוקת המפתחות. כאשר שרת רוצה לייצר רשומה חדשה הוא פונה לשרת המרכזי ודורש ממנו מפתח ייחודי, השרת המרכזי מכיר את המפתחות הקיימים בכל המערכת ולכן ידע להנפיק מפתח ייחודי.
   יתרונות: אין כפילויות של מפתחות.
   חסרונות:
   - עלויות תקשורת.
   - בקריסת השרת המרכזי, לא ניתן לבצע הוספת רשומות למערכת.
   - לא מתאים למערכת מבוזרת גדולה (השרת מהווה צוואר בקבוק).
2. Aliases – גישה זו מהווה פתרון לבעיות השיטה הקודמת. נוסיף טווח ערכים, או שימוש בתחילית הייחודית לכל שרת.
   יתרון: כל שרת יכול לייצר לעצמו מפתחות.
   חסרון: השקיפות נהרסת, בשליפת רשומה ניתן לראות שהמערכת לא באמת אחידה (רואים לפי התחילית מאיפה הרשומה).
   פתרון: נייצר פונקציית hash ונסווה את המיקום בו נוצר המפתח.

פתרון #2:

Transaction Mechanisms
נרצה לשמור על ACID במערכת מבוזרת. קשה לשמר את עקרונות אלו בגלל פעולות הintra (פעולה מסוימת יכולה לרוץ במקביל בין שרתים) וinter (פעולות של טרנזקציה מסוימת מתבצעות במקביל בין שרתים).

במערכת מבוזרת כל שרת מכיל 2 רכיבים: **Transaction Coordinator** ו**-Transaction Manager**.
TC – "שר החוץ" – הרכיב אליו מגיעה הטרנזקציה. הוא אחראי לנהל אותה מול שרתים אחרים ויתקשר עם השרתים האחרים.
TM – "שר הפנים" – מנהל את הטרנזקציה המתרחשת אצלו לוקלית, עבור פעולות שהתקבלו משרתים אחרים (אם TC של שרת כלשהו פנה לטרנזקציה שלו, הוא מנהל את ביצועה בתוך השרת).
חסרונות:
- שמירה על עקביות.
- במקרה של קריסה של הTC האחראי לא יהיה ניתן לתקשר עם השרת הנוכחי.
- במקרה של קריסה של הTM – אי אפשר להשלים את פעולת השרת ולכן גם לא את הטרנזקציה.



מה יכול להיות בעייתי בתהליך הזה?
1. קריסה של אחד השרתים.
2. מידע שמשוכפל מספר פעמים – איך נשמור על העקביות של כל העותקים האלו?

שמירה על עקביות ושלמות המידע – Concurrency Control

שמירה על העקביות באמצעות **מנעולים**. ישנם 2 סוגים של מנעולים עבור קריאה וכתיבה. כשמעדכנים ערך לא נרצה שמישהו יבצע במקביל קריאה/כתיבה.

סוגי מנעולים:

Shared Lock – עבור קריאה. מספר משתמשים יכולים לקרוא בו זמנית אבל הפריט נעול עבור כתיבה.

Exclusive Lock – עבור כתיבה. כשנבצע כתיבה לא נאפשר לאף משתמש לכתוב/לקרוא במקביל.

גישות לביצוע הנעילה במערכת מבוזרת:

Single Lock Manager – שרת מרכזי האחראי להקצות את כל המנעולים. אם אין נעילה אקסקלוסיבית על הפריט, השרת המרכזי נותן לשרת המבקש מנעול על פריט המידע.

יתרון: נוחות – שרת אחד שמנהל את הכל ומודע לכל מה שקורה. ***לא מגיעים למצב של deadlock.***

חסרון: כל העומס על שרת יחיד, אם השרת המרכזי קורס לא ניתן לקבל מנעולים.

Distributed Lock Manager – כל שרת מנהל מנגנון נעילה משל עצמו, **בהנחה שאין שכפול מידע בשרתים.**

אם טרנזקציה רוצה לבצע קריאה או כתיבה של משתנה, היא תיגש לשרת הרלוונטי ותבקש ממנו מנעול.

יתרונות:

- אם שרת אחד קורס השאר יכולים להמשיך לתפקד (כל מה שלא קשור למידע שנמצא בשרת שקרס). אין שרת שהוא נק' כשל.
- חלוקת עומסים בין השרתים.

חסרונות:

- במצב של שכפול מידע זה לא יעבוד.
- עלולים להיווצר Deadlocks. הקושי העיקרי יהיה לזהות את הDeadlocks כי כל שרת מנהל את הנעילות של עצמו.
- דרושה תקשורת רבה.
- דרושה אי תלות בין הטרנזקציות **Isolation** – טרנזקציה אחת לא משפיעה על הערכים של טרנזקציה אחרת.

גישות להתמודדות עם Replication ו-Distributed Lock Manager

1. **Primary Copy:** לכל רשומה קיים עותק ראשי בשרת כלשהו ועותקים בשרתים אחרים. כאשר טרנזקציה רוצה לבצע נעילה לרשומה, היא תיגש לעותק הראשי שלה ואם מתאפשר היא תקבל מפתח (בשהprimary copy מקבל נעילה הוא שולח באופן סמוי לכל העותקים הודעה על כך).

   יתרונות: מערכת עקבית, איזון העומסים בין השרתים.

   חסרון: אם העותק הראשי של הרשומה נופל, אין גישה אל הרשומה.
2. **Majority Locking:** כאשר טרנזקציה רוצה לבצע נעילה לרשומה, היא צריכה שרוב השרתים שמחזיקים אותה יסכימו לנעילה.

   הרוב נקבע ע"י הreplication factor (כמות העותקים של הרשומה). אם הרוב אישרו, הטרנזקציה תקבל נעילה. כך נשמרת העקביות כיוון שאף טרנזקציה אחרת לא יכולה לקבל נעילה ברגע שאחת קיבלה נעילה (כי היא כבר לא תוכל לקבל את הרוב).

   יתרונות: עקביות נשמרת, אם שרת אחד נופל עדיין ניתן לגשת לפריט.

   חסרונות: יש צורך בהרבה תקשורת, עלולים להיווצר Deadlocks (למשל בהינתן 3 טרנזקציות ו9=rf, אם אף אחת מהטרנזקציות לא קיבלה 5 אישורים ייווצר deadlock). ***בנוסף בעיה של עקביות – אם חלק מהשרתים קרסו, בעת "חזרתם לחיים" הם יחזרו עם ערך שאינו מעודכן.*** שיטה זו אינה מומלצת לשימוש כאשר יש הרבה קריאות משרתים.

   כאשר מתבצע עדכון על רשומה ניתן לבחור באחת מ2 דרכים:

   a. עדכון הרשומה בכל המופעים שלה בכל השרתים.
   b. עדכון הרשומה רק בשרתים שאישרו נעילה לאותה הטרנזקציה – אם נבחר באופציה זו, תהיה בעיה של עקביות. את בעיית העקביות נוכל לפתור ע"י שימוש בtime stamp – בעת עדכון הרשומה נשמור את זמן ביצוע העדכון, כאשר נרצה לשלוף את הרשומה נבחר בערך העדכני ביותר. כיוון שרוב השרתים כן מעודכנים (בזכות עקרון הרוב), ברגע שנשווה בין 2

רשומות שאחת יותר מעודכנת מהשנייה נבחר ברשומה המעודכנת, מפני שזו בהכרח הרשומה המעודכנת ביותר (לפי עקרון הרוב).

3. **Biased Locking:** מבחינים בין מנעולים לקריאה ולכתיבה. נותן מענה לבעיה הקודמת, שהרי יש בדר"כ יותר קריאות מכתיבות.
כאשר טרנזקציה רוצה לבצע כתיבה, היא צריכה לקבל exclusive lock מכל השרתים שמחזיקים אותה. כאשר טרנזקציה רוצה לבצע קריאה, היא צריכה לקבל shared lock משרת אחד.
חסרונות: אם שרת אחד קורס, לא ניתן לבצע כתיבה.
יתרונות: העקביות נשמרת בצורה יותר טובה מהשיטה הקודמת, כיוון שכל השרתים צריכים לאשר את הנעילות.
4. **Weak Consistency (transaction consistent):** לכל רשומה קיים primary copy שמנהל אותה.
כתיבה יכולה להתבצע רק דרך primary copy וקריאה יכולה להתבצע דרך כל שרת המכיל עותק.
שיטה זו מתאימה למערכות מתקדמות המעדיפות זמינות על עקביות.
חסרון: העקביות לא נשמרת. בעת עדכון של הprimary copy יכולה להתבצע קריאה במקביל, אך רק לאחר העדכון של הprimary copy יתבצע עדכון בשאר השרתים.
יתרון: אין deadlocks.
על מנת להפחית את הפגיעה בעקביות נאפשר **transaction consistent** – טרנזקציה מתחילה במצב עולם ידוע ולפיו היא צריכה גם לסיים. כלומר, טרנזקציה תושלם רק אם היא עוסקת בערכים שהתעדכנו לפניה. אם ערך התעדכן לאחר תחילת פעולת הטרנזקציה, היא תבוטל (כך נשמור גם על isolation).
(יתרון נוסף) ההסתברות לכך ששתי טרנזקציות יעדכנו את אותו פריט באותו זמן נמוכה יחסית, אך אם זה קורה, נבטל את הטרנזקציה (כנראה ההסבר למה אין deadlocks...).

לסיכום, מבחינת דרישות של Commit, נרצה לשמור על 2 עקרונות:
Atomicity – אם כל השרתים השלימו את פעולתם (הטרנזקציה הסתיימה בהצלחה בכל השרתים) נבצע commit, אחרת rollback (גם אם רק אחת נכשלה).
Durability – אם טרנזקציה בוצעה, השינויים ישתקפו בDB.

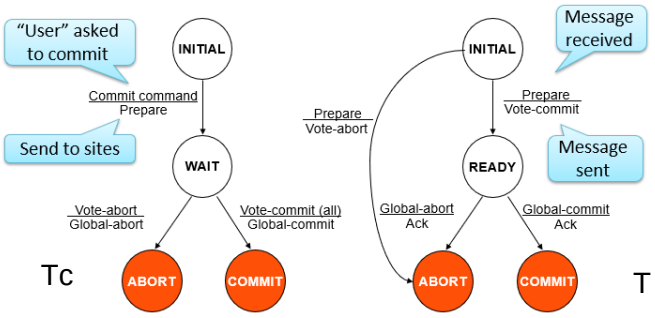
2 Phase Commit

מנגנון המנסה לממש commit גלובלי.

Centralized 2PC:
- הTc שולח הודעה לכל השרתים (לTm) האם הם מוכנים לביצוע. (Phase1)
- כל שרת מחזיר תשובה האם הוא מוכן או לא (<abort> או <ok>).
- אם כולם החזירו לTc תשובה חיובית, הוא שולח לכולם שיבצעו commit, אחרת שיבצעו **abort (אם משתתף לא הגיב)**. (Phase2)
- נשלחת אל הTc הודעת אישור הנקראת ACK, כשהוא מקבל מכולם אישור שהטרנזקציה בוצעה, היא תיכתב לLog. ***(זהו COMMIT מלא)***

Linear 2PC:
- Phase1 מתבצע רק עבור שרת יחיד וכך כל השרתים מעבירים אחד לשני את ההודעה בצורה סדרתית.
- גישה זו דורשת פחות תעבורה ברשת אך היא יותר איטית.

Distributed 2PC:
- Tc שולח לכולם את ההודעה של Phase1. ה-Tc אחראי רק על ייזום התהליך.
- המשתתפים שולחים אחד לשני האם הוא יכול לבצע commit או לא, במקום לשלוח בחזרה לTc כמו בריכוזי. כל אחד מהמשתתפים שולח לכל אחד מהמשתתפים האחרים האם הצביע בעד commit או בעד abort. ברגע שכל משתמש קיבל את כל ההודעות מכולם, הוא מקבל את ההחלטה בעצמו.
- אם לפחות שרת אחד החזיר לשרת כלשהו תשובה שלילית, הוא מחליט לבצע abort, אחרת יבצע commit.
- גישה בזבזנית מבחינת תעבורה אך **יותר מהירה.**



<u>מצבים:</u>

:Tc

Initial – שליחת הודעה לכל השרתים.

Wait – מחכה לתשובות מהשרתים.

Commit/Abort – שליחת ביטול/אישור לטרנזקציות.

:Tm

Ready – ביצוע השאילתה נעשה, רק מחכה לאישור לcommit.

Commit/Abort –ביטול/אישור הטרנזקציה.

Termination Protocol – כאשר משתתף לא מקבל תגובה מהTC, כאשר משתתף לא מגיב לTC.

<u>התמודדות עם כשלים בגישה הריכוזית:</u>

1. **קריסה של Tc:**

- קריסה בשלב הInitial (בשלב שליחת ההודעה) - לא בעיה.

- קריסה במצב בו המשתתפים בready – השרתים לא מקבלים הודעות ולכן תקועים עד שהTc חוזר לחיים. זו נקודת הכשל המרכזית. הם מחזיקים מנעולים ומחכים לפקודת Commit או Abort.

קריסה בשלב הAbort/Commit – לא בעיה.

התאוששות מקריסת Tc:

נחליף אותו בTc חדש או שנחכה שיתאושש. המשתתפים מחכים עד להחלפת/התאוששות הTc.

2. **קריסה של משתתף:**

- קריסת משתתף בשלב בו הTc נמצא במצב wait – הTc ימשיך לחכות לאישור מהמשתתף שקרס ויתקע.

- קריסת משתתף במצב של abort – לא נקבע אישור על ביצוע הabort אבל זה לא קריטי, כשהוא יחזור לחיים הוא יבטל אותה.

התאוששות מקריסת משתתף:

כשהמשתתף יחזור לחיים הTc יצטרך להחליט מה על המשתתף לעשות, לפי מדיניות שתוחלט.

כלומר, המשתתף מתאושש ומחכה להודעה מה-TC.

הגבלות של 2PC:

- Blocking – אם הTC קורס כשמשתתף ב-ready – אז המשתתף נחסם עד שהTC מתאושש.

- Blocking מורידה את הזמינות.

- ייתכנו deadlocks.

3PC: כאשר הTc קרס והמשתתפים נמצאים בשלב ready – זוהי המגבלה העיקרית של 2PC, מנגנון 3PC נועד לפתור את בעיה זו – לעיתים מונע blocking.

NoSQL

לבסיסי נתונים לא רלציוניים יתרונות רבים. למה עדיין מעדיפים את הרלציוניים?

1. יותר קל להישאר במשהו מוכר (כסף, משאבים).
2. לפעמים אופן שמירת הנתונים בNoSQL לא מאפשר לבצע שאילתות מורכבות.
3. שמירת העקביות – יש ארגונים שלא מוכנים "להקריב" עקרון זה ולכן חייבים להמשיך להשתמש בDB רלציוניים.
4. לא לכל הארגונים יש עיסוק עם Big Data. לחברות קטנות אין צורך במעבר הזה.

רעיון הNoSQL – המידע מבוזר על פני מספר שרתים **בצורה לא טבלאית**, לא מובטח שעקרונות הACID יסופקו.

<u>מאפייני גישת NoSQL:</u>
ביצוע Scaling: הגדלת כמות המידע בה ניתן לטפל בזמן נתון, יכולת קריטית. הScaling דורש מאיתנו הגדלת כמות שרתים וכו'.
מעדיפים יעילות על עקביות: העקביות בבסיסי נתונים לא רלציוניים פחות משמעותית. מצד אחד נרצה לשמור על אילוצים שהגדרנו ומצד שני לשמור על אחידות המידע בין כל השרתים המחזיקים את פריט המידע. הרבה פעמים נבחר לוותר על העקביות (כיוון שזמן הריצה של שנדרש כדי לממש הוא ארוך יחסית) ונתעדף הכנסות מהירות לDB.
Loose data model: מגדירים סכמה יחסית פשוטה וכשנעדכן מידע "נשפוך" את המידע לDB כפי שהוא ללא בדיקות נוספות.

נרצה לבצע Scale out – לאפשר למערכת לטפל ביותר פעולות, לשמור על אלסטיות וזמינות המערכת,impedance mismatch (העניין עם הOOP וה-DB).

עם הזמן פירשו את הNoSQL כ– not only SQL. כלומר שליבת הרעיון הNo SQL לא בהכרח בא לידי ביטוי בשפת השאילתות, אלא באופן שבו המידע ממודל כדי לענות על הצרכים הארגוניים (למשל הכנסה/שליפה מהירה וכו').
ה-DB הטבלאיים לא מתחרים עם ה-DB הלא טבלאיים, אלא מדובר ב-2 פתרונות שונים שמתאימים לארגונים שונים.

Scalability עבור אלפי/מיליוני שרתים – נתקשה לממש את עקרונות הACID (במיוחד את העקביות).
נעבור להשתמש ב-NoSQL (עם עקרונות מעט שונים).

<u>תיאוריית CAP</u>
בבסיסי נתונים מבוזרים, ניתן להשיג רק 2 מתוך 3 העקרונות הבאים: Consistency, Availability, Partition Tolerance.

Consistency: המשתמש יקבל את המידע העדכני ביותר (ערך שנעשה עליו commit או ערך עדכני יותר של טרנזקציה שלא עשתה commit אבל שינתה את ערך המידע). מובטח לנו שעבור טרנזקציה שהחלה בזמן X נשלוף ערך שנכון לכל הפחות בזמן X.
Availability: אם הוגשה בקשה לשרת – חייב לחזור חיווי למשתמש, כאשר החיווי יכול להיות גם שלא ניתן לבצע את הפעולה כרגע.
Partition Tolerance: המערכת תהיה מושבתת רק אם כל השרתים בה קרסו או אם קיימת תקלת תקשורת קולקטיבית במערכת. כלומר, מערכת יודעת להמשיך לעבוד גם במצב בו חלק מהשרתים במערכת קרסו ובמקרי קריסה המערכת יודעת לזהות אותם ולטפל בהם ולאחר תיקון התקלה, תחזור למצב המקורי.
בהכרח יקרו תקלות במערכת מבוזרת כי יש המון שרתים ובהכרח יש אחוז מסוים של שרתים שיקרסו, לכן עקרון זה חשוב מאוד.

Consistency + Availability: בשילוב זה בהכרח קיים עותק יחיד מכל פריט מידע (כי צריך להיות זמינים ועקביים) ולכן שילוב זה אינו ישים במערכות No SQL. אם המערכת מושבתת באופן כללי זה לא ישים.
כל מערכת NoSQL שמכבדת את עצמה תמיד צריכה לממש partition tolerance, כי תקלות של קצר בתקשורת/קריסה של שרתים וכו' צריכות להיתמך ע"י המערכת.

לכן, במערכות No SQL נבחר בין A+P לבין C+P.

Consistency + Partition Tolerance: הזמינות נפגעת במקרה של תקלה, כי נחכה עד לעדכון הערך הנדרש.

Availability + Partition Tolerance: החלוקה גורמת לצורך בתקשורת טובה בין השרתים, במקרה של תקלה נעדיף להחזיר ערך באופן מידי (גם אם אינו מעודכן), מתוך העדפת הזמינות על פני העקביות.

<u>בחירה בזמינות:</u>
- פתרון פשוט – הרבה פעמים נעדיף לתת תגובה מהירה למשתמש על פני עדכון הערך.
- רוב השאילתות מבצעות קריאה ולא כתיבה ולכן נעדיף לתת זמינות להרבה שאילתות קריאה מאשר עקביות למעט שאילתות כתיבה.
- שיקול עסקי – קל יותר לטפל בדברים בדיעבד מאשר ב-real time (למשל overbooking).

- (סיבה מהעולם האמיתי) רוב המידע שאנחנו אוגרים בDB גם ככה "מלוכלך" (בעייתי ולא עקבי) לכן רעש נוסף לא משנה.

BASE

אלטרנטיבה ל-ACID:
Basically Available: המערכת בכללותה צריכה להמשיך לתת מענה למשתמשים (בזמן קריסה).
Soft State: מצב המערכת והנתונים יכולים להשתנות לאורך זמן.
Eventual Consistency: לאחר פרק זמן נתון, המערכת תהיה עקבית (לאחר עדכון יש חלון זמן עם חסם עליון בו המערכת לא עקבית).

ACID אל מול BASE:
בACID אנחנו מניחים את התרחיש הגרוע ביותר. בBASE לעומת זאת, אנחנו מניחים תרחיש אופטימי.
בנוסף ACID היא מערכת מורכבת יותר וBASE פשוטה יחסית לניהול (עקביות חזקה וזמינות נמוכה אל מול עקביות חלשה וזמינות חשובה).
Base מבטיח לעשות base effort לכך שיהיה מידע עקבי – מאמץ בסיסי.

ACID:
- Strong consistency.
- Less availability.
- Pessimistic concurrency.
- Complex.

BASE:
- Availability is the most important thing. Willing to sacrifice for this (CAP).
- Weaker consistency (Eventual).
- Best effort.
- Simple and fast.
- Optimistic.

2 סוגי עקביות:
Strong consistency – בעת ביצוע עדכון, זה ישתקף בכל השרתים המכילים עותק באופן מידי.
Weak consistency – לא מובטח שיתבצע עדכון בכל השרתים מיד, אלא מתישהו. חלון הזמן בין עדכון הפריט לבין עדכון העותקים נקרא **Inconsistency Window**.

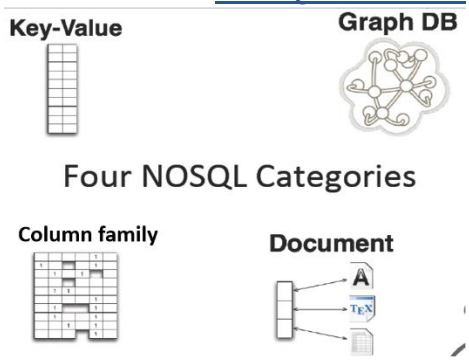
Eventual Consistency: צורה מיוחדת של weak consistency. אם התבצע עדכון לפריט מסוים, לאחר זמן t כלשהו, כל השרתים יכילו את הערך המעודכן של פריט זה. אם לא התרחשה תקלה במערכת, נוכל לתת חסם עליון ל**Inconsistency Window** (אחרת לא נדע כמה זמן התקלה תימשך). חלון זמן זה תלוי ב2 דברים:

1. ***Replication Factor – מספר השכפולים של הפריט במערכת.***
2. זמני התקשרות בין השרתים השונים.
3. ***מדיניות ה-read/write של הארכיטקטורה המבוזרת.***

<u>גרסאות של **Eventual Consistency**:</u>

1. Read your writes consistency: מבטיח שטרנזקציה תקרא את הערך העדכני ביותר שהטרנזקציה כתבה (העקביות בתוך הטרנזקציה נשמרת).
2. Monotonic Read consistency: אם טרנזקציה ביצעה קריאה לפריט X בזמן t ולאחר מכן קוראת אותו שוב, לא יקרה מצב שקראה ערך שעודכן לפני t (או שאקרא את מה שקראתי קודם, או ערך עדכני יותר).
3. Monotonic Write consistency: אם טרנזקציה כתבה ערך לפריט, אז בעתיד בקריאת הפריט, נקרא את הערך שעדכנו ולא ערך ישן יותר.

NoSQL Variants



ארבעת הסוגים שונים זה מזה בעיקר באופן שבו המידע נשמר.

Key-Value

שומרת את המידע במודל של מיפוי key-value, כמו מילון.

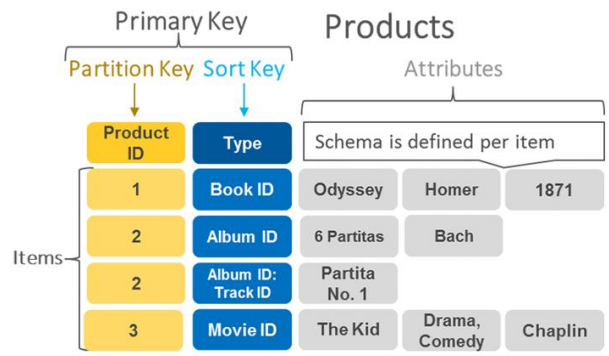


Table => Items => Attributes

הגדרת מפתח וערך כאשר ערך הוא רצף שדות, לכל מפתח ייתכן כמות שדות שונה. מפתח יכול להכיל שדה יחיד או מפתח ראשי ומשני. מפתח ראשי יכול להיות מורכב מPartition key וSort key.
Partition Key: משמש להכנסה או לשליפה, איתו נגיע לpartition הרלוונטי באמצעות פונקציית Hash (מיקום פיזי).
Sort Key: לאחר הגעה לpartition המתאים, ניעזר בSort Key כדי למצוא את הרשומה הרלוונטית.

יתרונות: הכנסה ושליפה מהירה (לפי מפתח), מודל פשוט, horizontal scaling.
חסרונות: קשה למדל data מורכב, פשטני מדי.

DB מוכרים: DynamoDB, Redis.

דוגמה – DynamoDB:

Partition key= hash attribute
Sort key = range attribue

```
var params = {
    TableName : "Music",
    KeySchema: [
        { AttributeName: "Artist", KeyType: "HASH" },   //Partition key
        { AttributeName: "SongTitle", KeyType: "RANGE" }   //Sort key
    ],
```

Redis

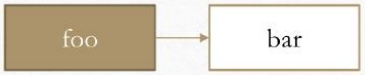
ה-DB מאוחסן ב-RAM, אך עדיין מאפשר גיבויים ויציבות.

Jedis – ספריית JAVA עבור Redis.

```
Jedis jedis = new Jedis("132.72.65.45");
jedis.set("foo", "bar");
String value = jedis.get("foo");
jedis.close();
System.out.println(value);
```

bar

- Connecting from java



תומך במבני נתונים שונים:

```
jedis.set("events/city/rome", "32,15,223,828");
String cachedResponse = jedis.get("events/city/rome");
```

The variable *cachedResponse* will hold the value *"32,15,223,828"*. - String

```
jedis.lpush("queue#tasks", "firstTask");
jedis.lpush("queue#tasks", "secondTask");

String task = jedis.rpop("queue#tasks");
```

The variable *task* will hold the value *firstTask*. - List

Redis Data Structures Cont.

- **Sets**: an unordered collection of Strings.

```
jedis.sadd("nicknames", "nickname#1");
jedis.sadd("nicknames", "nickname#2");
jedis.sadd("nicknames", "nickname#1");

Set<String> nicknames = jedis.smembers("nicknames");
boolean exists = jedis.sismember("nicknames", "nickname#1");
```

The Java Set nicknames will have a size of 2, the second addition of nickname#1 was ignored.

exists variable will have a value of *true*.
The method *sismember* enables you to quickly check for the existence of a particular member.

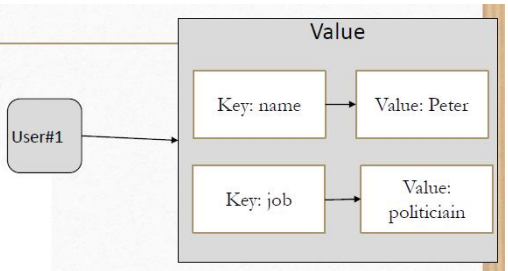
- Set

- **Hashes**: mapping between *String* fields and *Object* values:

```
jedis.hset("user#1", "name", "Peter");
jedis.hset("user#1", "job", "politician");

String name = jedis.hget("user#1", "name");

Map<String, String> fields = jedis.hgetAll("user#1");
String job = fields.get("job");
```



- Hash

- **Sorted Sets:** Set where each member has an associated ranking.

```java
Map<String, Double> scores = new HashMap<>();

scores.put("PlayerOne", 3000.0);
scores.put("PlayerTwo", 1500.0);
scores.put("PlayerThree", 8200.0);

scores.keySet().forEach(player -> {
    jedis.zadd("ranking", scores.get(player), player);
});

String player = jedis.zrevrange("ranking", 0, 1).iterator().next();
long rank = jedis.zrevrank("ranking", "PlayerOne");
```

The variable *player* will hold the value *PlayerThree* because we are retrieving the top 1 player and he is the one with the highest score. .

The *rank* variable will have a value of 1 because *PlayerOne* is the second in the ranking and the ranking is zero-based.

- Sorted Set

הסבר:

- zadd(key, score, member) – יצירת sorted set של ranking לפי ה-score, כך שה- key=player# נהפך ל-value בסט הממוין החדש.
- zrevrange(key, start, stop) – מחזיר value (member).
- zrevrank(key, member) – מחזיר את ה-score (שהוא בעצם ה-key של ה-sorted set).

טרנזקציות: מבטיחות אטומיות. כלומר, בקשות ממשתמשים אחרים לא יטופלו באופן מקבילי בזמן שטרנזקציה רצה ב-Redis.

```java
String friendsPrefix = "friends#";
String userOneId = "4352523";
String userTwoId = "5552321";

Transaction t = jedis.multi();
t.sadd(friendsPrefix + userOneId, userTwoId);
t.sadd(friendsPrefix + userTwoId, userOneId);
t.exec();
```

key-value

Column Family

בטבלאות הרשומות יש מספר עמודות משתנה.

DB מוכרים: BigTable (גוגל), Cassandra (פייסבוק), HBase (אפאצ'י).
נדגים על BigTable:

- טבלה מאגדת סביב ישות.
- Tablet: סט של שורות (range partition). לכל tablet נגדיר column family.
- Column Family: קבוצת עמודות. מספר הColumn families הוא קבוע, אבל מספר העמודות משתנה ממשפחה למשפחה.
- מספר העמודות משתנה מרשומה לרשומה.
- נשים לב שמתבצעות הוספות בלבד – בעת עדכון ערך, נוסיף ערך חדש ולא נמחק את הערך הקודם (על מנת לתמוך בהכנסה מהירה). לכל עדכון תהיה חותמת זמן ובשליפה יישלף העדכון האחרון.

2 Column Families

2 cols 2 cols

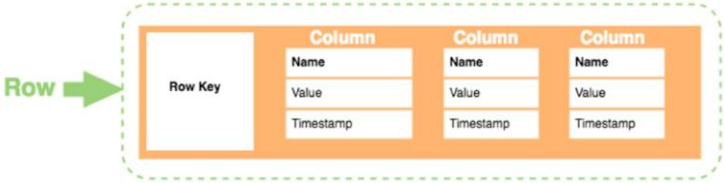
Row Key	cf: personal_data		cf: professional_data	
empid	name	city	designation	salary
1	alice	new york	manager	50,000
2	bob	san francisco	sr. engineer	30,000
3	cindy	seattle	jr. engineer	25,000

יתרונות: תומך במידע מורכב יחסית, אינדקסים טבעיים (עמודות), טוב בScaling אופקי.
חסרונות: קשה למדל מערכת מאוד מורכבת.

Cassandra

מודל Wide Column Key Pair.

Wide Column: מיפוי רב ממדי. מבוסס על עמודות. טבלאות עם מספר עמודות לא קבוע. הוספה לעמודות עם חותמת זמן (ולא מחיקת ערכים קודמים).
יתרונות: ***מודל יותר מובנה***, תומך באינדקסים. חסרון: לא תומך ב-JOIN.



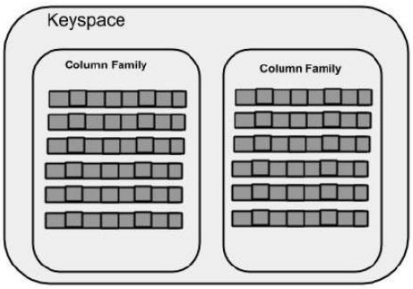
Cassandra **– יתרונות:** מאוד scalable, מאפשר להוסיף חומרה. מתוכנן לנהל כמויות גדולות של מידע. מאפשר זמינות גבוהה ללא נק' כשל אחת. ניתן להשתמש עם RDBMS.

היררכיית הארכיטקטורה:
Cluster: רכיב המכיל data center אחד או יותר.
Data center: אוסף של racks הקשורים אחד לשני.
Rack: אוסף של nodes הקשורים אחד לשני.
Node: המקום בו מאוחסן המידע. כל node הוא עצמאי אך מקושר לnodes אחרים. כל node ב-cluster יכול לקבל בקשות קריאה/כתיבה, לא משנה איפה המידע מאוחסן ב-cluster. אם node נופל, אז הבקשות יטופלו ע"י nodes אחרים ברשת.

Gossip Protocol: תקשורת בין nodes כדי לזהות כאלה לא תקינים בcluster, ע"י החלפת state messages – כך לומדים בצורה מהירה על מצבם של nodes אחרים ב-cluster.

Read Consistency levels: Cassandra מתעדפת זמינות על עקביות לפי רמות. (יש הבדל בין עקביות של קריאה לבין של כתיבה)
ONE: זמינות גבוהה, עקביות נמוכה – תשובה חייבת לחזור רק מ-node אחד המכיל עותק של מידע.
QUORUM: תשובה חייבת לחזור מ $1+\frac{n}{2}$ nodes ***(ערך תחתון)*** כפול ה-replication factor, כלומר **מרוב של עותקים**, לא משנה מאיזה data centers.
Local QUORUM : תשובה חייבת לחזור מ $1+\frac{n}{2}$ nodes (***ערך תחתון, רוב***) שמכילים עותקים ונמצאים באותו data center.
All: עקביות גבוהה (תשובה מכולם), זמינות נמוכה.

Data Model:
Keyspace: שם כולל ל-Column families הקיימים.
Column Family: אוסף מסודר של שורות. כל שורה היא אוסף מסודר של עמודות.
Column: מבנה הנתונים הבסיסי ביותר – 3 ערכים: מפתח שמי או שם עמודה, ערך, חותמת זמן.



CQL: ממשק המאפשר גישה ל-CASSANDRA, חלופה ל-SQL.

דוגמה למרכיבי שאילתה: SELECT col_name1..col_name100 FROM col_family WHERE key=key_value

Syntax:

```
INSERT INTO <tablename>
(<column1 name>, <column2 name>....)
VALUES (<value1>, <value2>....)
USING <option>
```

```
INSERT INTO student (student_id, student_fees, student_name)
VALUES(1,5000, 'Ajeet');
INSERT INTO student (student_id, student_fees, student_name)
VALUES(2,3000, 'Kanchan');
INSERT INTO student (student_id, student_fees, student_name)
VALUES(3, 2000, 'Shivani');
```

- **<option> is:**
  - CONSISTENCY <consistency_level>
  - TTL <seconds> - TTL column values are automatically marked as after the requested amount of time has expired.
  - TIMESTAMP <integer>

- Insert

```
public static void pouplateData() throws SQLException {

    String data=
            "BEGIN BATCH \n"+
                "insert into news (key, category2, linkcounts,url) values ('user1','news',75,'news.com') \n"+
                "insert into news (key, category, linkcounts,url) values ('user2','tech',15,'tech.com') \n"+
                "insert into news (key, category, linkcounts,url) values ('user3','travle',415,'ba.com') \n"+
                "insert into news (key, category, linkcounts,url) values ('user4','search',45,'goog.com') \n"+
            "APPLY BATCH;";

    PreparedStatement st = con.prepareStatement(data);
    st.executeUpdate();
    st.close();
}
```

- ע"י batch

```
DELETE [ column_name [ , column_name ] [ … ] | column_name [ term ] ]
FROM [ keyspace_name. ] table_name
[ USING TIMESTAMP timestamp_value ]
WHERE row_specification
```

```
DELETE firstname, lastname FROM cycling.cyclist_name WHERE firstname = 'Alex';
```

```
DELETE FROM cycling.cyclist_name WHERE firstname IN ('Alex', 'Marianne');
```

To delete more than one row, use the keyword IN and supply a list of values in parentheses, separated by commas:

```
DELETE firstname, lastname
  FROM cycling.cyclist_name
  USING TIMESTAMP 1318452291034
  WHERE lastname = 'VOS';
```

The TIMESTAMP is an integer representing microseconds. You can identify the column for deletion using TIMESTAMP.

- Delete

1. SELECT * from People;
2. SELECT Name, Occupation FROM People WHERE key IN (199, 200, 207);
3. SELECT FIRST 3 REVERSED 'time199'..'time100' FROM Events;
4. SELECT COUNT(*) FROM users;
5. SELECT COUNT(*) FROM big_columnfamily LIMIT 20000000;
6. SELECT * from People USING CONSISTENCY QUORUM;

- Select

Java: חיבור - בדומה ל-Oracle. עבודה עם statements ו-result set.

Document DB

מיפוי מפתחות ל-document. Document הוא סוג של מבנה נתונים – מסמך של תגיות key-value למשל XML/JSON.
ישנו אוסף של מסמכים כאשר כל מסמך מייצג ישות – כמו רשומה בטבלה. מעין collection מסוים של מסמכים המקביל לטבלה בעולם הרלציוני.

MongoDB: הפעולות מתבצעות באופן יעיל, תמיכה באגרגציות, יצירת אינדקסים.
יתרונות: DB מאוד גמיש (ללא סכמה), Scaling טוב אך לא מצוין (ככל שלDB יותר יכולות, כך הScaling נפגע).
חסרונות: היכולת למדל תלויות בין ישויות ומידע מקושר היא מוגבלת. השאילתות מוגבלות לפי key ואינדקסים.

Graph DB

ממדלים את המידע באמצעות גרף בעל קודקודים וצלעות. הקודקודים יכולים להכיל תכונות מסוימות וגם אפשר לתת תכונות על גבי הצלעות. קשתות בין קודקודים מייצגות קשר בין הקודקודים האלו, כלומר ממדלות את היחסים בין האובייקטים.

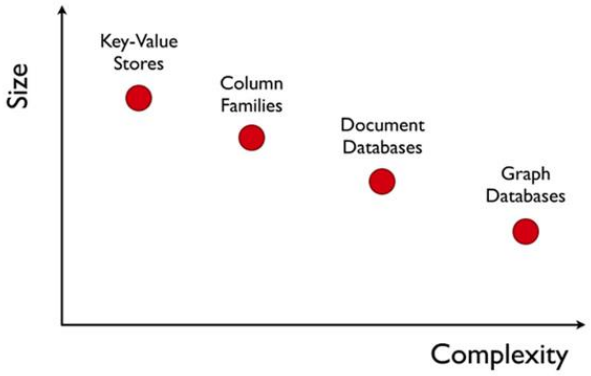
בהינתן גרף G וביטוי רגולרי r:
regular path queries: מציאת כל זוגות הקודקודים (x,y) בG כך שיש נתיב מx לy שמספק את r.
Reachability query: האם קיים מסלול מקודקוד מסוים לקודקוד מסוים.

Neo4J: פחות Scalable, לא תומך בהרבה שפות תכנות, ניתן לתמוך בקריטריונים של ACID (בניגוד לאחרים) ותומך בטרנזקציות.

מדוע להשתמש בDB מסוג זה?
- מערכת המלצה: המלצות למשתמשים בהתאם לצרכים שלהם או בהתאם לפרופיל הנבנה עליהם.
- זיהוי הונאות באמצעות גרפים.
יתרונות: מידול תלויות וקשרים, DB מהיר (אלגוריתמי גרפים ממומשים בצורה יעילה).
חסרונות: פחות סקלבילי.

NoSQL and Scalability

בשני ממדים: Data Size, Data Complexity.



ניתן לראות כי יש tradeoff בין היכולת למדל מידע מורכב לבין גודל המידע. תמיד יש trade-off בין רמת המורכבות לבין ה-size ולכן צריך להבין איפה הנקודות האופטימאלית עבורנו/עבור המשתמש ולפי זה לבחור את ה-DB שמתאים לנו.

	Performance	Scalability	Flexibility	Complexity	Functionality
Key-Value Stores	high	high	high	none	variable (none)
Column stores	high	high	moderate	low	minimal
Document stores	high	variable (high)	high	low	variable (low)
Graph databases	variable	variable	high	high	graph theory
Relational databases	variable	variable	low	moderate	relational algebra

איך לבחור NoSQL DBMS?

- Data model:
איך נרצה לאחסן לגשת לdata?
סכמה דינמית?
- Storage: בזיכרון או בדיסק?
- API: SQL, Java...
- עקביות:
עקרונות CAP,
עקביות משפיעה על מהירות קריאה וכתיבה.
- נעילות ו-deadlocks:
האם לאפשר גישה בו-זמנית למספר משתמשים?
האם משתמשים בנעילות?
האם אין deadlocks?

	RDBMS	Most NoSQL
Popularity	Very High	Increasing
Tools	Many	Few
Consistency	ACID	Limited
Query Execution	Rich, Fast	Limited
Standard	SQL	No Standard
Scalability	High	Very High
Schema	Static	Dynamic

- התאמה לקריאה/כתיבה.
- ניהול תקלות:
  איך?
  האם ניתן להמשיך להתנהל למרות התקלות?
- דחיסה?
- Load balancing:
  האם המערכת מצליחה לאזן עומסים?

Hadoop

פלטפורמה המאפשרת לאחסן ולעבד מידע בצורה יעילה ומבוזרת. Open source.
מורכבת מ2 מרכיבים עיקריים:
HDFS – מערכת הקבצים שמאפשרת לנהל אותם בצורה מבוזרת ומאחסנת אותם. ***Write once, read many.***
Map Reduce – מאפשרת לעבד פרטי מידע, לבצע טרנספורמציות ועוד.

Commodity Clusters: שרתים שנועדו לעבודה בצורה מבוזרת. נעדיף כמות גדולה של שרתים (בחירה בכמות על פני איכות) ונעדיף להשקיע בתקשורת בין השרתים, נשתדל לצמצם את כמות התעבורה למרחקים ונשתדל שהתעבורה תהיה בעיקר בין שרתים קרובים.

Distributed File System

הקבצים יכולים להיות בגדלים שונים והם מחולקים לchunks, כל chunk משוכפל 3 פעמים (ברירת מחדל) וגודל כל chunk הוא לרוב 128Mb. master – מנהל שיודע איפה כל chunk שמור.
Nodes – שרתים בודדים.
Rack – ארון של שרתים.
Cluster – אוסף של racks.

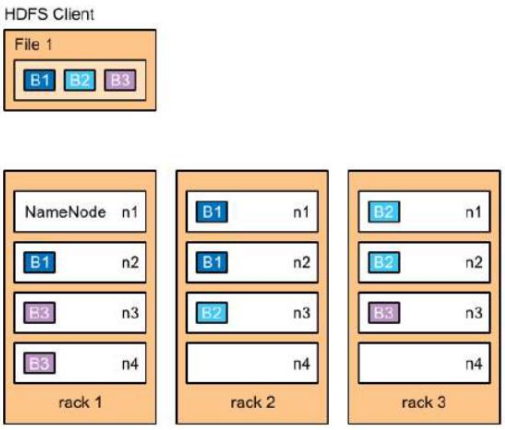
השרתים עובדים בארכיטקטורה של Shared Nothing.

Switch – דרך התקשורת בין השרתים (גם nodes וגם racks).

מהירות התקשורת: בין שרתים באותו הrack – 1 Gbps, בין 2 racks שונים – 2-10 Gbps. הקצב מהירות טוב יותר ב2 עד 10 כי הrack גם מעביר מידע בין השרתים וגם הוא משמש לתקשורת עם שאר ה data centers.

כתיבת קובץ לHDFS

תהליך אחסון קובץ חדש המגיע למערכת, בהנחה שקיימים 3 racks במערכת שלנו –
ראשית, נחלק את הקובץ ל3 חלקים (chunks), כל chunk משוכפל 3 פעמים ונשמר באופן הבא: נפנה לmaster של אחד הracks **(הקרוב ביותר ל-client)**, בrack אחד נשמרים 2 עותקים וב-rack נוסף (שונה מהראשון) נשמר העותק *השלישי*. ***סה"כ שמירה ב-2 racks.***
לאחר שמירת כל עותק ניגש לשרת הבא בו נשמור את העותק הבא, בסוף יוחזרו הודעות אישור באופן רקורסיבי עד לאישור הסופי לmaster ממנו החלה העתקת הchunk.
**** נכתוב קובץ תחילה לשרת הקרוב ביותר.***



- ***נשמור כל פיסת מידע ב2 racks שונים*** כיוון שכך אם rack מסוים קורס, נוכל לגשת לrack אחר ולשלוף ממנו את המידע.
- נשכפל את אותו פריט מידע פעמיים באותו rack – כי אם מגיעה משימה לrack המכיל 2 עותקים, 2 השרתים יכולים בו זמנית להפיץ את המידע של הchunk לשאר הracks (לזרז את התהליך).

איך לבחור איפה לכתוב רשומה?

2 גישות:

1. **Modulo based Hashing:** גישה בסיסית. בהינתן פריט מידע חדש המגיע, ניקח מזהה ייחודי של הchunk ונפעיל עליו פונקציית hash - modulo לפי כמות השרתים ולפי התוצאה נחליט איפה לשמור את פריט המידע.

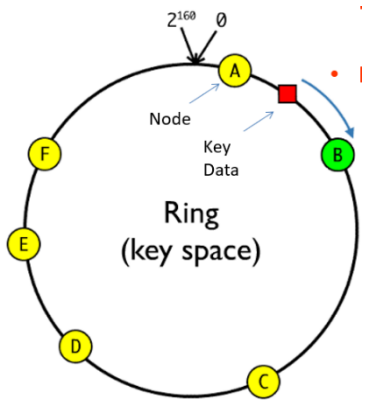
$$partition = ID\%num\_of\_nodes$$

חסרון: בהינתן ששרת קרס, פונקציית הhash שלנו משתנה, לכן אם שרת קורס נצטרך לסווג מחדש את כל פרטי המידע לשרתים.

פתרון לכך בגישה השנייה:

2. **Consistent Hashing:** כפתרון לשיטה הקודמת, מטרתנו תהיה לשנע במקרה קריסה **(הוספה/הסרה)** מספר מינימלי של קבצים. מבנה שמירת הקבצים הוא טבעתי.

השיטה: טווח הערכים של כלל השרתים הינו $[0,2^{160}]$, לכל שרת נקצה טווח ערכים מתוך טווח זה, כאשר הטווח המתקבל עבור השרת הוא מאוד גדול.
בהגעת קובץ חדש למערכת, נפעיל עליו את פונקציית ה*hash*, אם אין שרת בדיוק בערך זה, נקצה את הקובץ להישמר בשרת הבא בתור (שאחראי על טווח הערכים בו "נפל" הערך).
ה*Replication Factor* שלנו הוא 3 ולכן, נשמור בנוסף לעותק הראשוני שנשמר כפי שתיארנו, עותק שני ושלישי בשני השרתים הבאים בתור לפי כיוון השעון.

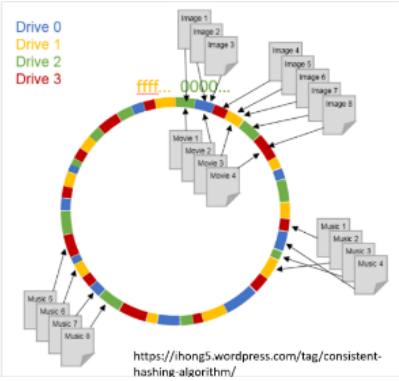


התמודדות במקרה קריסה: נניח שרת *B* קרס.

1. טיפול בערכים שמופו לשרת שקרס:
נתבונן על שרת *B*, כל הערכים שמופו אליו הם הערכים בין *A* לבין *B*. בהינתן ששרת *B* קרס, הערכים שמופו לטווח זה ישוכפלו לשרת אחר. נזכור כי יש לנו עותקים של שרת *B* ב*C* וב-*D* ולכן נוכל להעתיק את העותקים משם אל שרת *E*.
2. טיפול בערכים שנשמרו כעותק בשרת שקרס:
נשים לב ששרת *B* שמר את הערכים של שרת *F* כעותק (הערכים שמופו לטווח בין *E* ל*F*), ובנוסף גם את הערכים של שרת *A* כעותק. על מנת לשמור על ה*Replication Factor*, נצטרך לשכפל את פרטי המידע האלו ל*C* ול-*D* בהתאמה.

<u>איך נחלק את השרתים על פני הטווח?</u>

Virtual nodes - tokens: הגדרת תתי טווחים קטנים לכל שרת (node) ע"י virtual nodes – tokens, אשר ישמשו כיחידה עצמאית באותו שרת אך למעשה ישבו פיזית בשרת. כך נוכל לחלק את הטווח לכמות גדולה יותר וליצור חלוקה מאוזנת יותר.
לא נרצה לחלק את ה-*tokens* לפי רצף אחיד, כדי שבמקרה של קריסה העבודה תתחלק בין שאר השרתים השונים (ולא תמיד על אותו שרת לפי *node* מסוים שקרס).
באופן זה, ננצל בצורה מיטבית את המשאבים שיש לנו, נוריד את העומס על השרתים ובמקרה קריסה עבודת השכפול תקרה במספר חלקים במקביל.



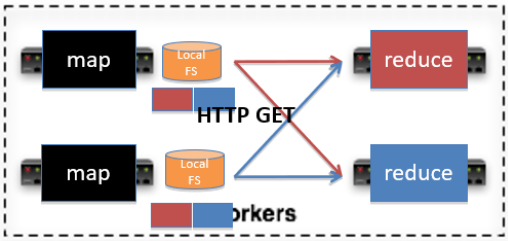
Map – Reduce

מערכת המאפשרת לעבד מידע בצורה מבוזרת. פלטפורמה לעיבוד מידע בנפח גדול. מערכת זו היא שקופה למשתמש במימוש שלה והיכולת לביצוע Scaling מאוד טובה.
Map: קריאת המידע וביצוע טרנספורמציה בסיסית עליו.
Reduce: קבלת המידע שעבר טרנספורמציה וביצוע פעולות עליו.
בין שתי פעולות אלה יש שלב ביניים שנקרא **Shuffle and Sort**, בו מחליטים לאן להפנות את התוצאות שהתקבלו מהפלט של map אל הreducer.
הmapper וה-reducer הם שרתים במערכת המבוזרת, כאשר כל שרת מבצע משימת map או reduce אחרת.

Map-Reduce Algorithms: פעולת ה-map מסדרת את הקלט בצורה של key ו-value. הפלט של משימות ה-map יהיה ממוין לפי ה-key, כך שכל מפתח מסווג למשימת reduce. פעולת ה-reduce מחברת ערכים המשויכים למפתח מסוים.

Dataflow in Hadoop

הmaster הוא מנהל הפעולות בcluster, הmaster קיבל משימה מהuser. הmaster יודע איפה המידע הנדרש יושב והוא זה שמקצה משימות לnodes השונים בcluster.



השרתים ייקראו workers, הmaster יחליט מי מהשרתים הם mappers ומי reducers.

כל שרת מחולק כך שחלק ממנו מוקצה לHDFS, חלק למערכת קבצים לוקלית וחלק ממנו מוקצה למשימות.

HDFS	Local FS	Map/Reduce

1. כל שרת (יכולה mapper מקבל בלוק של מידע מהHDFS להיות מוקצית לו יותר ממשימה יחידה).
2. כל mapper מעבד את המידע ולאחר העיבוד שומר אותו בLocal FS בזוגות של key וvalue (key הוא לא בהכרח ייחודי).
3. פעולת Shuffle and sort מעבירה את כל הזוגות בעלי אותו key לreducer המתאים להם.
4. הreducer מבצע את פעולתו (חישובים שונים).
5. לאחר שסיים, הreducer כותב את התוצאות למערכת הקבצים הכללית - HDFS.

<u>הערות:</u>
- שרת אחד יכול לקבל יותר מפעולת map/reduce אחת כיוון שכך נוכל לחסוך זמני בטלה של שרתים והתאוששת מתקלה תהיה מהירה יותר.
- בעיקרון ניתן להתחיל את פעולת הreduce לפני סיום הmap, ***אבל ב-Hadoop מכריחים את הmappers לסיים לפני תחילת הreduce.***

<u>הגדרה פורמלית:</u>
מבנה בסיסי זוג של key וvalue.

$$map: (key, val) \rightarrow < (k_1, v_1), (k_2, v_2), \dots, (k_n, v_n) >$$
$$reduce: (k', < v_1', v_2', \dots, v_n' >) \rightarrow < (k', v_1''), (k', v_2'') \dots >$$

Master and Worker

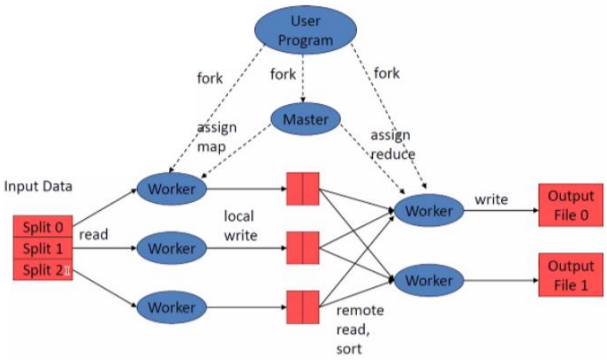
מתקבלות M משימות של mappers וR משימות של reducers. הmaster מחליט איזה workers יוקצו לmap ואיזה לreduce. לפי שיקולים שונים, כדי להחליט לאיזה worker להקצות איזו משימה, נרצה גם לדעת את המיקום והמרחק בין הworker למידע.

שיקולים:

- יעשה מאמץ למקם mappers בקרבת ה-nodes המכילים את בלוקי ה-HDFS בהם שמורים הקבצים אותם עליהם לנתח.

קביעת גודל M וR:

- M וR יהיו גדולים מכמות השרתים בCluster – מקטין את זמן ההתאוששות, מאפשר pipelining, איזון עומסים.
- **M>R** – ככל שיהיו יותר mappers הפעולה תהיה יותר ממוקבלת – כי הmappers מבצעים מיון וה-reducers מבצעים פעולות אגרגציה לכן הפעולה הקריטית היא של הmap. וגם בסוף מתקבלים R קבצי פלט – עדיף מעט.



כל mapper מייצר R קבצים מקומיים הנשמרים בFS מקומי המכילים זוגות (key,value) – כמספר פעולות הreduce (R).

* Intermediate keys – אותם זוגות הkey-value שנוצרו בשלב הביניים ע"י הmappers.

shuffle and sort: ע"י shuffle מועבר המידע מהmappers אל הreducers המתאימים ו-sort אחראי על מיון המפתחות בתוך הreducer ומסייעת לreducer להבחין בין המשימות השונות.

Coordination:
-הסנכרון שהmaster צריך לקיים בין כל הworkers שעובדים במערכת על משימה. כדי לבצע תיאום, עלינו לשמור לכל משימה שקיבלנו מהמשתמש, מה הסטטוס שלה (בתהליך/מחכה להרצה/הושלמה).
- כשהreducers כותבים לHDFS את הפלט הם מודיעים לmaster שהם סיימו ואז הmaster מקבל את כתובות שמירת הקבצים ומשם ישלוף את המידע הרלוונטי למשתמש.
- הmaster מוודא שהworkers עדיין בחיים ע"י הודעות ping – נקודה קריטית להשלמת המשימה. אם שרת קרס ניתן את המשימה שהוקצתה לו לשרת אחר.

Master – תחומי אחריות:

1. חלוקת המשימות של map וreduce לworkers.
2. בדיקה ווידוא שהworkers בחיים.
3. הmaster אחראי להודיע לreducers איפה הקבצים שהmappers שמרו הרלוונטיים עליו.
4. אחראי להודיע לuser על סיום הפעולה ולשלוף את מיקום התוצאה.

Communication from Map to Reduce: נשתמש בפונקציית hash, נשלח את הפונקציה לכל הmappers והיא תחלק את הזוגות שנוצרו בכל mapper לכמות הR שהגדרנו. פונקציית הhash תחזיר מספר בין 1 לR וכך כל זוג ימופה לreducer המתאים לו.

K-Means

Clustering:
supervised learning – למידה באמצעות מידע מתויג.
unsupervised learning – למידה ללא מידע מתויג.

סיווג – האלמנטים באותה הקבוצה דומים זה לזה ובקבוצות שונות, שונים זה מזה.

דמיון – משתנה לפי הגדרות שלנו - תלוי במה אנחנו רוצים לחפש/להדגיש בנתונים. מוגדר על פי Similarity function.

קיימים 2 סוגי clustering:

1. Partitional – חלוקת המידע לפי קטגוריה.
2. Hierarchical – חלוקת המידע בצורה היררכית ואז קובעים איפה החלוקה "נחתכת".

K-means Algorithm: אלגוריתם מסוג Partitional.
בהינתן dataset עם דגימות בעלות תכונות מסוימות נגדיר מספר k, האלגוריתם יחזיר k קבוצות.

1. ראשית נגדיר מרכזים המוגדרים באופן רנדומלי.
2. נשלוף את הנקודות מהdataset.
3. עבור כל דגימה, נבדוק את המרכז הקרוב אליה ביותר ונשייך את הנקודה אליו.
4. "נצבע" את הדגימות לפי המרכזים.
5. נגדיר מרכז חדש לכל cluster ע"י ממוצע הדגימות שהשתייכו לאותו הcluster.
6. נחזור לשלב 3 עד אשר לא מתבצע שינוי משמעותי במרכזים/נקודות לא משנות את השיוך שלהן/עד למספר איטרציות שנגדיר מראש.

חסרונות:
- לא מובטח פתרון אופטימלי וגם קיים קושי בתיוג פתרון אופטימלי.
- זמן ריצה גדול באופן תיאורטי (מעשית ההתכנסות מהירה יחסית).

K-means Issues:
- אתחול אקראי גורם לקבלת clusters שונים.
- כשהמידע אינו נומרי נצטרך לבצע המרה כלשהי.
- נצטרך להגדיר מראש מהו הk – k לא מתאים יגרום לסיווג לא מוצלח.
- דגימות רועשות משפיעות על האלגוריתם.

מימוש K-means עם Map Reduce:

1. המידע ההתחלתי נשמר בHDFS (בצורה מבוזרת).
   הdataset שמכיל את הדגימות מוקצה לmappers ול-reducers.
2. כל mapper מקבל את המרכזים הראשוניים וחלק מהדגימות. (**Mapper input: k centroids, samples**)
   כל mapper יחשב את המרחק האוקלידי של כל דגימה מהמרכזים שקיבל, כפלט יחזיר לכל דגימה את המרכז אליו הייתה הכי קרובה. (**Mapper output: <key=centroid, value=sample>**)
3. הreducer מקבל את הoutput של הmapper ויבצע עבור כל הנקודות שהשתייכו לcentroid מסוים את הממוצע של כלל הדגימות, כך למעשה יחושבו המרכזים החדשים.
   (**Reducer input: <key=centroid, value= [sample1, sample5...]**>
   את המרכזים החדשים יכתוב הreducer לHDFS.

* נשים לב שבכל איטרציה המידע נכתב ונקרא מחדש מ ואל הHDFS ולכן כל mapper יצטרך לקרוא את אותה החתיכה של דגימות.

Combiners: מטרת הcombiner היא צמצום כמות המידע המועברת בין הmappers לreducers. שלב זה מתבצע אחרי הmapper, נבצע אגרגציה לכל הזוגות עם אותו מפתח ונאחדם לזוג.

<u>שימוש בK-means:</u>

1. mapper מסוים ישייך מספר דגימות לאותו המרכז.
2. באמצעות הcombiner נבצע ממוצע של כל הדגימות המשויכות לאותו מרכז בmapper. מכיוון שמדובר בממוצע, נצטרך לשמור גם את מספר הדגימות עליהן הממוצע נעשה.
3. הreducer יבצע ממוצע משוקלל בהתחשב במספר הדגימות המצורף לכל חישוב.
4. הreducer כותב את המרכזים החדשים לHDFS.

כך נוכל לחסוך שליחת מידע מיותר מהmapper לreducer, המידע המועבר הוא כפונקציה של מספר המרכזים במקום מספר הדגימות (מספר משמעותית קטן יותר).

* נוכל להפעיל combiner רק כאשר הפעולה שתישלח היא קומוטטיבית או אסוציאטיבית, אחרת הreducer יתבסס על חישובים לא נכונים.

Relational Algebra Operations באמצעות Map-Reduce

Projection

נבצע את פעולת הprojection באופן הבא:
קלט לmapper: רשומות מהטבלה <r,r>.
נשלוף את השדות הרלוונטיים מכל רשומה.
פלט מהmapper: tuples של רשומות <r',r'>. כלומר tuple של שדות רלוונטיים מתוך כל רשומה. (הkey והvalue זהים).
קלט לreducer: <r',[r',r',…,r']>.
הreducer יטפל בכפילויות שעלולות להיווצר (עבור שדות שאינם מפתח).
פלט מהreducer: <r',r'>.

* פעולה זו נתמכת גם עבור מידע שאינו מסודר בצורה רלציונית.
* בעיה: פעילות של פעולה זו מוגבלת בפעולות הI/O שיש לבצע.
* המידע בHDFS נשמר כמחרוזת (שמירה כמחרוזת היא מהירה), לעומת שליפה המתרחשת לעיתים נדירות.

Selection

שליפת רשומות לפי תנאי כלשהו.
קלט לmapper: התנאי וסט של רשומות (יכולים להיות מספר תנאים שיופיעו בתנאי כחיתוך של תנאים).
פלט מהmapper: <r,r> אם הרשומה עונה על התנאי, אחרת אין פלט.
אין צורך בreducer בפעול זו, כי למעשה קיבלנו מהmapper את התשובה לcondition שרצינו (אלא אם נרצה לבצע פעולות נוספות שלא קשורות לselection – למשל group by, sort).

Union, Intersection, Difference

Union: מיזוג בין 2 טבלאות.
קלט לmapper: רשומה r.
פלט מהmapper: <r,r>.
קלט לreducer: <r,r>.
האיחוד מתבצע בreducer והטבלה המאוחדת חוזרת ללא כפילויות.

Intersection: החזרת רשומות שמופיעות ב2 הטבלאות. הקלט והפלט לmappers כמו בUnion.
קלט לreducer: <r,[r,..,r]>.
הreducer יוציא כפלט רק את הרשומות שברשימה שלהם מופיע יותר מזוג אחד (נחזיר את הkey שלהם).
אם קיימת כפילות של רשומות באותה הטבלה, נשמור בערך של הזוג את שם הטבלה ממנה הגיע הזוג וכך הreducer יבצע חיתוך רק לטבלאות שונות.

Difference: בהינתן R\S נרצה להחזיר את הרשומות שמופיעות בR ולא בS.
קלט לmapper: רשומה r.
פלט מהmapper: <r, table> כלומר המפתח זה הרשומה והvalue זה הטבלה ממנה הגיעה הרשומה.
קלט לreducer: <r, [table1, table2...]>
הreducer יבצע את הפעולה הבאה:
- עבור קלט מהסגנון <t,[R]> נחזיר <t,t>.
- עבור קלט מהסגנון <t,[S,R]> או <t,[R,S]> או <t,[S]>, לא נחזיר כלום.

Group By

קלט לmapper: רשומה r.
פלט מהmapper: <col, calculated_col> כלומר המפתח זה העמודה עליה נעשה group by וה-value זה השדה החישובי.
קלט לreducer: <r,[cc1,cc2,..]> כלומר השדה עליו נעשה group by כמפתח והשדות עליהם ייעשה החישוב כרשימת ערכים.
פלט מהreducer: < col, calculated_val>.

* ***ניתן לחשב זאת עם הcombiner.*** נבצע איחוד סביב מפתח ונחשב את הממוצע עבור הערך כבר בmapper ונעביר לreducer את הממוצע ואת מספר הערכים עליהם נעשה הממוצע.

Join

Reduce Side Join:

קלט לmapper: רשומות מהטבלה <r,r>.
פלט מהmapper: <join_column, r>, העמודה עליה נבצע join כמפתח והרשומה עצמה.
קלט לreducer: <col_val, [r1,r2,r3,..]>. ערך השדה עליו מתבצע הjoin ואוסף רשומות המשויכת לkey הזה.
הreducer יטפל בכפילויות שעלולות להיווצר (עבור שדות שאינם מפתח).
פלט מהreducer: תוצאת הJoin.
נשים לב, כל הרשומות שיש להן אותו ערך בשדה של הjoin יגיעו לאותו reducer (כיוון שהוא המפתח).

חסרון מרכזי: התעבורה הגדולה שעוברת בין הmappers לreducers היא יקרה ומאטה מאוד את התהליך.
הmapper לא מסנן את המידע ולכן כל המידע עובר בשלמותו לreducers (טבלאות מאוד גדולות).

Map side Join – Parallel scans:

גישה טובה ויעילה, אך פחות פרקטית בשל **ההנחות תחתן היא עובדת:**
- הטבלאות עליהן מתבצע הjoin חייבות להיות ממוינות לפי שדה הjoin.
- צריך לחלק את הטבלאות עליהן מתבצע הjoin לאותה כמות חלוקות.
- כל הרשומות בעלות אותו key צריכות להימצא באותו partition.

נגדיר את כמות הmappers שיעבדו וכל mapper יקבל חלוקה מ2 הטבלאות. בשל ההנחות, כל הרשומות באותו הmapper יהיו כל הרשומות שיצטרך לצורך ביצוע הJoin.
כך לא נצטרך להשתמש בreducer (כי הJoin התבצע בmapper).

<u>יתרון</u>: חיסכון בתעבורת נתונים בין הmapper לreducer.
<u>חסרונות:</u>
- בעייתי להסתמך על מספר רב של אילוצים.
- mapper יכול להכיל כמות רשומות עצומה אם הטבלה לא מאוזנת (המון רשומות עם אותו המפתח – צריכות להיכנס לאותו הmapper) *גם הreduce join דורש את זה אבל שם יש פתרונות שמאזנים את זה.

<u>In – Memory Join:</u>

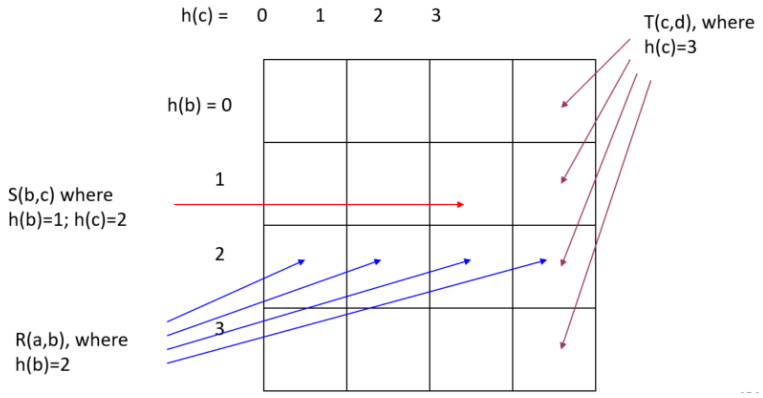
<u>הנחה מרכזית:</u> נניח שיש 2 טבלאות R וS כאשר אחת הטבלאות (R) קטנה מספיק על מנת להיכנס בשלמותה לזיכרון של כל mapper.
- נשלח את R לכל הmappers ולכל mapper נשלח חתיכה מS.
- עבור כל רשומה של S בכל Mapper נבדוק האם יש התאמה לR, אם כן נבצע join, אחרת לא (R משוכפלת לכל הmappers).

<u>1. Striped Variant:</u>
אם R לא יכולה להיכנס בשלמותה לזיכרון הmapper, אז נחלק אותה לn חלקים כך שכל חלק כן יכול להיכנס לזיכרון הmapper. כלומר בהינתן m חלקים של S וn חלקים של R נצטרך m*n mappers.

<u>2. Memcached Joined:</u>
הארכיטקטורה שלנו היא Shared Nothing. ניקח זיכרונות RAM ממספר שרתים שונים ונאחד אותם באופן וירטואלי לזיכרון משותף גדול אליו נוכל לטעון את הטבלאות.
המגבלה בשיטה זו היא התקשורת, נצטרך תקשורת טובה ומהירה בין השרתים. לכן אם התקשורת טובה השיטה הזו מעולה, אחרת התהליך מאבד ממשמעותו. לפעמים על מנת לפתור בעיה זו נעביר את המידע בbatch.

<u>Three Way Join:</u> אם נרצה לבצע join בין יותר מ2 טבלאות נבחר בשיטה זו. ביצוע join כזה עלול לקחת הרבה זמן כיוון שיש הרבה פעולות I/O.

1. **<u>Cascade of 2-way joins:</u>**
נבצע קודם join בין 2 טבלאות, נרשום את התוצאה בHDFS ואז נבצע join בין הטבלה הנותרת לבין הטבלה המאוחדת שכתבנו לHDFS.
2. **<u>Mapping for 3-way join:</u>**
נתבונן בjoin הבא: $R(A, B) \bowtie S(B, C) \bowtie T(C, D)$.
נפצל כל טבלה לmappers, כל mapper יפעיל פונקציית hash על שדות הטבלה וישלח אותם לbucket המתאים להם.
נמפה כל tuple בS הנראה כך S(b, c) (S הטבלה בעלת השדות המשותפים ל2 הטבלאות) באופן הבא:
key: (h(B), h(C)), value: (S, B, C) – הרשומה עצמה.
נמפה כל tuple בR ל1 עד k הbuckets שיש לנו.
key: (h(B), y) כך ש $1 \le y \le k$. value: (R, A, B) – הרשומה עצמה.
נמפה כל tuple בT ל1 עד k הbuckets שיש לנו.
key: (x, h(C)) כך ש $1 \le x \le k$. value: (T, C, D) – הרשומה עצמה.
נקבל מטריצה ריבועית כך שלכל רשומה בS יש תא ספציפי במטריצה, לכל רשומה בR יש מופעים במטריצה עבור כל העמודות בשורה h(B) ולכל רשומה בT יש מופעים עבור כל השורות בעמודה h(C).
כלומר, כל reducer מקבל סט רשומות שמתאים לתא הספציפי לפי הhash.



Fault Tolerance Via Re-Execution

Map Reduce הוא מאוד fault tolerance. הוא מתמודד עם קצר בתקשורת, תקלות בתוכנה וכו'.

התמודדות עם כישלון של worker: ראשית, זיהוי התקלה בCluster יתבצע ע"י הmaster, שאחד מהתפקידים שלו הוא לוודא במשך כל הזמן שכל אחד מהשרתים עדיין פועל (ע"י כך שהוא שולח להם הודעות שהם צריכים להגיב אליהן, אם הן לא מגיבים אחרי כמה זמן הוא מסיק שקרתה תקלה ואז אם הם היו צריכים להריץ איזושהי משימה של הmap-reduce היא לא תוכל להיות מושלמת אם השרת קרס. לכן הוא יקצה את המשימה הזאת לשרת אחר).

התמודדות עם כישלון של master: קיים master יחיד שאחראי לבצע הקצאות ומכאן שאם הוא קורס, המערכת "תקועה" – לכן הMaster מהווה את נקודת הכשל שלנו. אם master כשל, Hadoop צריך לוודא הריגה של הmaster שכשל כדי שלא יחזור לחיים ויפריע לmaster החדש שנקצה.

הבדלים בין Map Reduce לבין Parallel DBMS

- Map-reduce זוהי טכניקה לעיבוד מידע ואילו DB רלציוני מאחסן מידע.
- DB מבוזר רלציוני תומך בסכמה, בניגוד לmap reduce.
  Hadoop שומר את המידע במחרוזות, ***בדיקות התקינות למידע יעשו רק בשליפתו*** (הכנסה מהירה ושליפה איטית יותר).
- אינדקסים לא קיימים בmap reduce.
- בDB רלציוני התכנות הוא בSQL לעומת תכנות בSpark או Java.
- אופטימיזציה אינה אפשרית בmap reduce.
- גמישות וfault tolerance אפשריים בMap reduce ופחות בDB.
- מבחינת שרתים בDB יש הרבה שרתים אבל בMap reduce יש 10K של commodity servers.

	Parallel DBMS	MapReduce
Schema Support	✓	Not out of the box
Indexing	✓	Not out of the box
Programming Model	Declarative (SQL)	Imperative (C/C++, Java, ...) **Extensions through Pig and Hive**
Optimizations (Compression, Query Optimization)	✓	Not out of the box
Flexibility	Not out of the box	✓
Fault Tolerance	Coarse grained techniques	✓
Number of Nodes	A few Large Nodes	10k of commodity servers

מגבלות של Map Reduce

1. I/O Intensive – בפעולות map reduce מתבזבז המון זמן על קריאה וכתיבה מהדיסק (פעולה יקרה).
2. כתיבה קשה – כתיבה בצורה של map reduce היא כתיבה קשה ולא נוחה למשתמש.
3. לא מתאים לעיבוד תהליכים מורכבים.
4. לא כל בעיה ניתן להגדיר עם map reduce.
5. לא מתאים למערכות המצריכות מענה מידי (למשל streaming).
6. לא מתאים לשאילתות מובנות.

עם השנים התפתחו כלים מבוססי Hadoop המפצים על הבעיות של map reduce:

Apache Pig

פלטפורמה לניתוח נתונים (***גם גולמיים***) בצורה מהירה המבוסס סקריפטים (פעולות קצרות המתבצעות בצורה מהירה). עובדת על גבי Hadoop. ניתן לכתוב פקודות בשפה שדומה לשפתו תכנות מוכרות – pig Latin. הscript שכותבים עובר למנוע של pig שיתרגם את ה-pig Latin script לאוסף של משימות של map-reduce (map-reduce statements שמתבצעות על גבי הפלטפורמה).
יש בpig רכיב שמבצע אופטימיזציה ועוזר למתכנת להתמקד בקוד שלו ולא בייעול עבור פעולות map-reduce.

Pig Latin Script – נותן מענה על מגבלת הכתיבה הקשה של Map Reduce.

יתרונות:
- מאפשר להשתמש באופרטורים נוחים: join, sort וכו'.
- שפה פשוטה לתכנות.
- מתבצעת אופטימיזציה לסקריפטים שהמשתמש כתב והוא לא צריך לדאוג לביצוע יעיל של השאילתות.
- ניתן להגדיר פונקציות מותאמות אישית.
- תמיכה בכל סוגי הdata (כי גם Hadoop יודע לתמוך בכל הסוגים).
- ניתן לשמור משתנים.
- מספק יכולות לביצוע ETL.

בהשוואה ל-Map-Reduce: בניגוד לmap reduce העבודה עם pig היא בhigh level. הוא כלי מונחה מידע (data flow) – המשתמש אומר מאיפה לקרוא את המידע ולאן לשמור אותו. יש פעולות (כמו join) שאי אפשר לבצע בmap reduce אבל אפשר בpig.

בהשוואה ל-SQL: בSQL המידע הוא טבלאי וב-Pig אינו מוגדר. האופטימיזציה בpig היא ברמה נמוכה יותר מאשר SQL. בpig ניתן להוריש טבלה וב-SQL זה לא אפשרי.

```
input_lines = LOAD '/tmp/my-copy-of-all-pages-on-internet' AS (line:chararray);

-- Extract words from each line and put them into a pig bag
-- datatype, then flatten the bag to get one word on each row
words = FOREACH input_lines GENERATE FLATTEN(TOKENIZE(line)) AS word;

-- filter out any words that are just white spaces
filtered_words = FILTER words BY word MATCHES '\\w+';

-- create a group for each word
word_groups = GROUP filtered_words BY word;

-- count the entries in each group
word_count = FOREACH word_groups GENERATE COUNT(filtered_words) AS count, group AS word;

-- order the records by count
ordered_word_count = ORDER word_count BY count DESC;
STORE ordered_word_count INTO '/tmp/number-of-words-on-internet';
```

Hive

מהווה גשר בין עולם הSQL לעולם המבוזר.

הוא סוג של מחסן נתונים לעיבוד נתונים ב-Hadoop (data warehouse – מערכת מידע שמרכזת מידע גולמי ומטרתה לחלץ ממנו תובנות לגבי המידע). מאפשר לנתח big data על גבי Hadoop בקלות.

Hive הוא לא:

- DB רלציוני (אך מובנה בטבלאות יחסית רופפות, אין מפתחות וכו').
- כלי שמתאים ל-online transaction processing (OLTP – עיבוד טרנזקציות בזמן אמת).

Hive הוא כן:

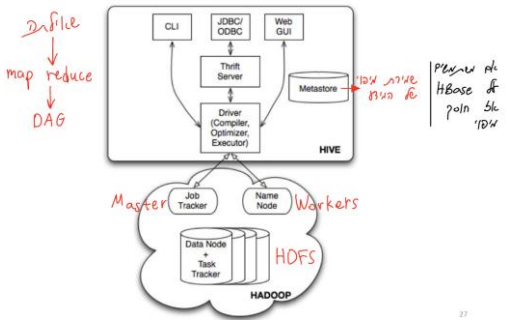
- כלי המאפשר לשמור מידע בHDFS.
- מותאם ל-online analytical processing (OLAP).
- מספק שפת שאילתות HQL.
- מהיר ו-scalable.

ניתן לשלב שאילתת HQL עם Map reduce.

אופן הפעולה באמצעות Hive:

- משתמש כותב שאילתת HQL.
- השאילתה מגיעה לקומפיילר שמרכיב גרף DAG (גרף מכוון חסר מעגלים) עם משימות המapreduce שחולצו מן השאילתה המקורית.
- לאחר מכן המידע נשלף מהHDFS.
- המשימות נשלחות בזו אחר זו אל המaster ומתבצעות.

Meta Store: רכיב המכיל מידע עבור כל טבלה. מכיל מיפוי של הטבלאות למיקומים השונים בHDFS, כלומר איזה עמודות יצרנו ומה מיקום העמודות בHDFS.



יכולות עיקריות:

- מאפשר לבצע אינדוקס.
- שומר כל סוג של מידע (text, HBase).

- ניתן לבנות user defined functions (UDF).
- **יכולות SQL במערכת מבוזרת** – אשר מתורגמות ל-map-reduce ומשימות Spark.

Yarn

בא לתת מענה על חוסר היכולת לביצוע חישובים מורכבים בMap Reduce.

כלי זה מאפשר לנהל את משאבי Hadoop. מצד אחד ינהל את המשאבים בצורה יעילה יותר מגרסתם הבסיסית בHadoop, **מצד שני מאפשר להריץ תוכנות שלא כתובות רק בצורה של Map reduce.**

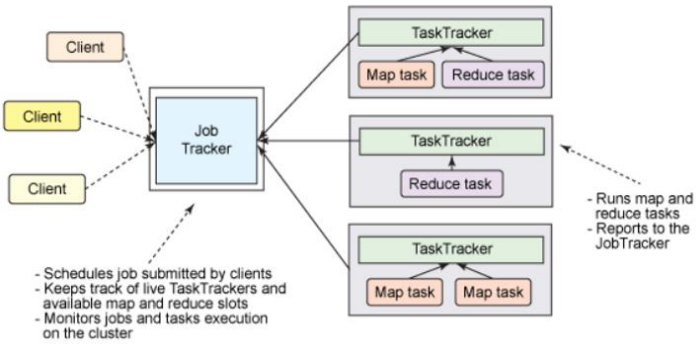
חלק מרכזי בגרסה 2.0 של Hadoop.

Hadoop v1.0

Job Tracker: זהו הMaster. במבנה זה האחריות על הMaster מאוד גדולה כי הוא מרכז הכל אצלו.
Task Tracker: בתוך כל שרת יש רכיב כזה. רכיב זה אחראי לדווח מה קורה בשרת שלו לMaster.

בגרסה זו, הקצאת המשאבים היא קשיחה ויוריסטית (כלומר לא דינאמית ולא אופטימלית - למשל אם חלק מהשרתים סיימו וחלק לא, לא יתאפשר לשרתים אחרים לעזור לאלה שסיימו).

<u>חסרון:</u> מפחית ביכולות של map-reduce.



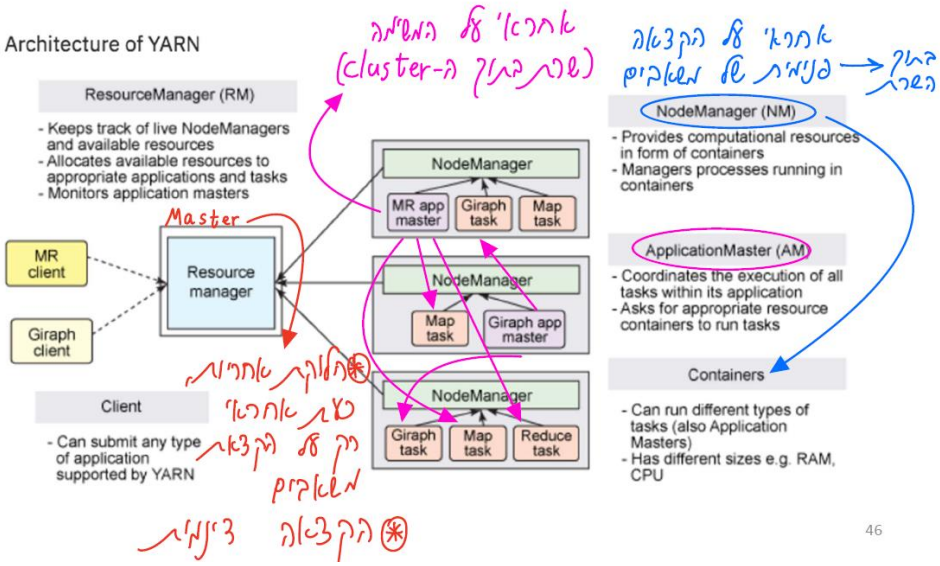
Hadoop v2.0 – Yarn Architecture

פתירת בעיה של הקצאת משאבים בצורה מהירה וגם מאפשר להריץ מספר טכניקות תכנות ולא רק map-reduce.

<u>מרכיבים:</u>

Resource Manager – מקבל משימות ומקצה משאבים (במקום הmaster). כאן ההקצאה היא דינמית, אם משימה דורשת יותר/פחות משאבים, ניתן לשנות זאת. כלומר הוא לא מתערב בהקצאה הפנימית של המשאבים בתוך השרתים.
Application Master – אחראי על משימה מסוימת. לראות שכל השרתים עובדים על המשימה. אם שרת כשל, הוא זה שצריך לנהל את המשימות שלו ולהעבירן לשרת אחר.
Node Manager – מנהל את התהליכים הנמצאים בתוך הNode הספציפי ב-containers. ***מאפשר הקצאת משאבים דינמית.***
Container – אוסף של מחיצות בתוך השרת המאפשרות למשימה לרוץ מבלי שתזלוג למשימות אחרות, הפרדה וירטואלית בין התהליכים השונים.

שלבי העבודה: הלקוח מגיש בקשה לResource manager והוא מאתחל באיזשהו שרת application master שאחראי על ניהול משימה זו. הapp master אחראי על ניהול המשא ומתן על משאבי המשימה אל מול הresource manager. לאחר הקצאת המשאבים, נקבל מהresource manager את מיפוי השרתים המתקבלים למשימה. כל שרת יבצע את משימתו וכשיסיים יודיע לapp master על סיום המשימה וזה יעלה עד ללקוח.



Spark

נותן מענה על כמות הפעולות הגדולה של I/O בHadoop.

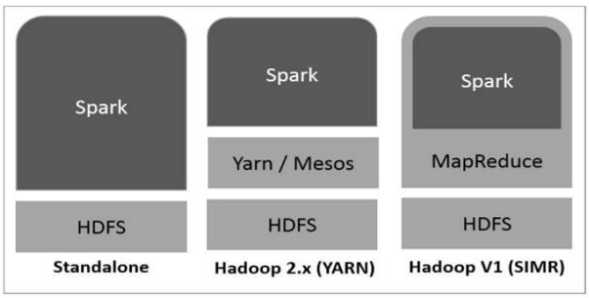
מנוע יותר עוצמתי מHadoop ומהיר יותר הנבנה לעיבוד מידע מקבילי. נבנה על גבי Hadoop ופועל על הHDFS.

Hadoop הוא כלי שפחות מתאים לפעולות בreal time אלא לbatch (אוסף של פעולות שנאגרות ביחד).
Spark תומך בנקודה הזו, הוא מנוע הרבה יותר מהיר שפועל ע"ג הHDFS.

<u>יכולות של Spark:</u>
- בנוי לפעולות Batch (הרבה מידע בבת אחת).
- מאפשר לעבר מידע המגיע בצורת Streaming.
- תומך בשאילתות SQL.
- מימוש של אלגוריתמים של Machine learning.
- open source, אחד הכי שמישים.

מאפיינים:
- Speed – Spark יכול להגיע למהירות של פי 100 מאשר Hadoop.
- תומך במספר שפות ותמציתי – תומך בjava, Scala, python.
- אנליזות שונות – ביצוע join, count וכו' (לא רק Map Reduce).
- מתאים ל-OLTP (עיבוד טרנזקציות בזמן אמת) – real time.

מימוש Spark בHadoop:
<u>Standalone</u>: Spark מחליף את מי שעיבד את המידע (את הmap reduce).
<u>Hadoop 2.x</u>: הSpark יושב על גבי הyarn ולא ניגש ישירות לHDFS.
<u>Hadoop V1</u>: בתוך הMap Reduce מגדירים את הSpark.



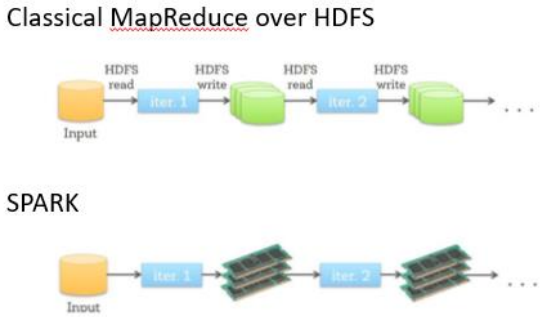
Spark Architecture

1. **RDD – resilient distributed dataset:** מבנה נתונים בסיסי בSpark. כשנבצע עיבוד בשאילתה נוכל להגדיר שהמידע ישמר בRDD.
   immutable, read only – הוא מוגדר להיות בלתי ניתן לשינוי. אם היה ניתן לשנות, היינו צריכים פרוטוקולים כדי לשמור על עקביות המערכת (כי הוא מבוזר).
   הוא in memory כלומר המידע מבוזר על הRAM של השרתים השונים – גורם לשליפה מהירה של הנתונים.
2. Core: מאפשר פעולות בצורה מהירה.
3. SparkSQL: מאפשר לבצע שאילתות SQL.
4. Streaming: כלי שמגיב לפעולות אינטראקטיביות (עיבוד מהיר יותר) – לא קיים בHadoop.
5. MLlib: ספריה הממשת אלגוריתמי Machine Learning.
6. Graphx: כלי המאפשר לנתח נתונים בצורה גרפית.

איך Spark נותן מענה לבעיית פעולות הI/O המרובות?

בצורה הרגילה של ביצוע map-reduce אנו קוראים מידע מהHDFS, מבצעים איטרציה אחת וכותבים את התוצאה לHDFS וחוזר חלילה – הרבה פעולות קריאה וכתיבה לדיסק.

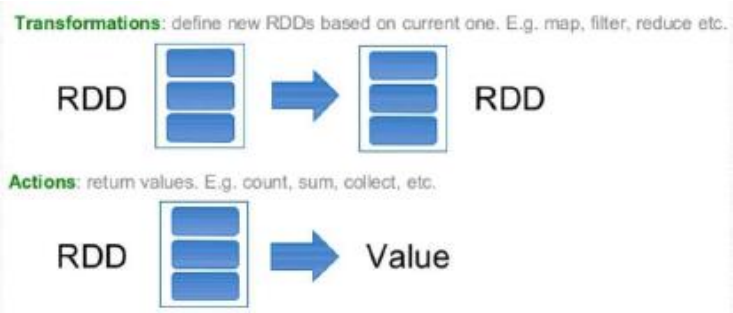
Spark מאפשר קריאה מהHDFS וכתיבה רק בסוף התהליך – באמצע אפשר לשנות: מבצעים איטרציה אחת ושומרים את המידע בRDD (סכמה מבוזרת שלא ניתנת לשינוי). באיטרציה השנייה אין צורך לקרוא את המידע מהדיסקים כי הוא שמור בmain memory. כך מבצעים עוד איטרציות ובסיום התוצאה הסופית נכתבת בHDFS. כך יש רק שתי גישות עיקריות לדיסק – בקריאה הראשונית ובכתיבה האחרונה.
במימושים של Spark בגלל שהמידע נכתב לRDD אז לא צריך בכל פעם לקרוא את המידע מהדיסקים, נוכל לקרוא אותו מהRAM.



Spark הוא Lazy

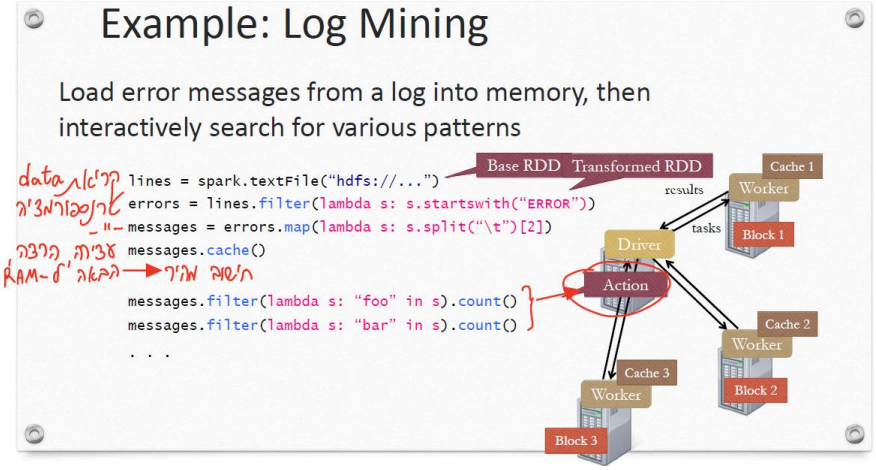
Spark לא מבצע חישובים של המידע מראש. עד שאין חישוב שנרצה לשלוף מהמערכת הוא לא יתבצע.
במקום שהSpark יבצע חישוב מסוים על כל הdata ואז המשתמש יבקש לשלוף רשומה ספציפית, מראש הSpark מחכה לבקשת המשתמש ולפיה מבצע חישוב על הרשומות הרלוונטיות.

<u>על כן, ישנן 2 סוגי פעולות בSpark:</u>
1) **Transformation** – מRDD לRDD (פעולות של map, filter, group by, reduce...).
2) **Actions** – מRDD לValue (פעולות של count, collect, save, sum...).



Spark Syntax and functions:

SparkContext: אובייקט החיבור לפונקציונליות של Apache Spark. מאפשר לאפליקציית Spark לגשת ל-Spark cluster.
Collect: הפיכת הRDD לרשימה של ערכים (איסוף כלל המידע המרכיב את הRDD מהשרתים בהם הוא מבוזר). נרצה להשתמש בפעולה זו לאחר טרנספורמציה כלשהי על הנתונים כדי לא לייבא כמויות מידע גדולות מדיי שיביאו לקריסה.
Parallelize: פעולה שלוקחת איזשהו מבנה נתונים (בדרך כלל בסגנון רשימה) והופכת אותו להיות RDD, מבזרת אותו על פני הזיכרון (RAM) של מספר שרתים שונים ולא מאפשרת לשנות את המידע שנמצא בתוכו (כלומר הופכת אותו ל-immutable).



שני חסרונות:

3. אי אפשר לבצע טרנזקציות עם שאילתות מסודרות (queries) כי המידע לא מבוזר בצורה מובנית – לכן החישובים לא מהירים.
4. Map-reduce היא פלטפורמה לא יעילה – מתאים יותר לפעולות batch ולא real time.

HBase

נותן מענה על כך שHadoop לא מותאם לביצוע שאילתות מסודרות – transactional DB over HDFS.

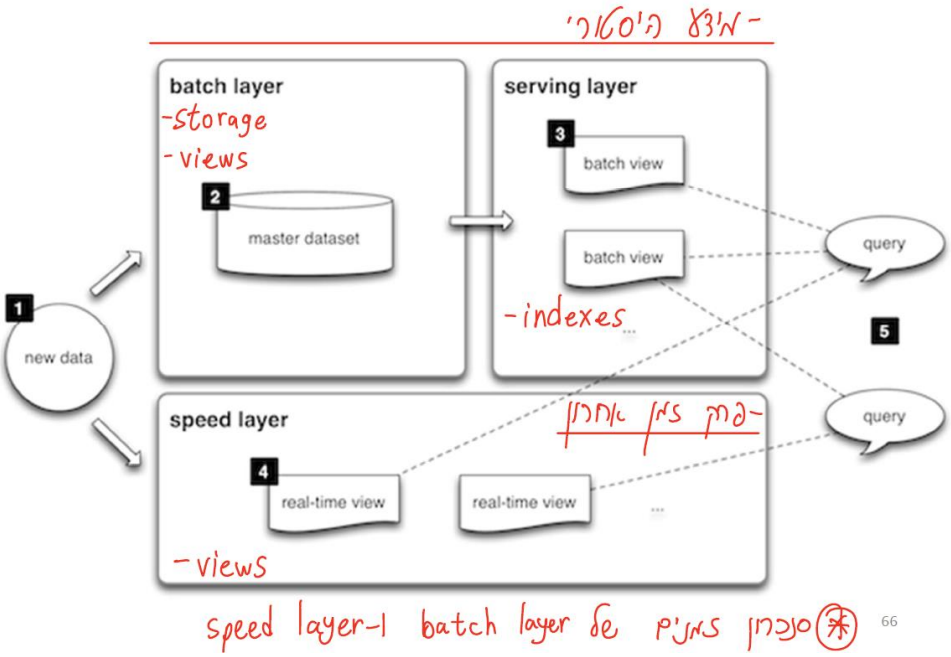
DB מסוג NoSQL המזכיר את Big Table מאפשר לשמור את הנתונים בצורה יותר סכמתית.
למעשה wide column DB.
Hive יכול לשבת על HBase ולעבוד איתו.

מאפיינים: קריאה/כתיבה בזמן אמת של big data, קריאה/כתיבה אטומית, טבלאות ענקיות.

Lambda Architecture

נותנת מענה על כך שHadoop לא מתאים לפעולות Realtime.

נרצה מצד אחד לאפשר לבצע שאילתות ב-real time ומצד שני לשמור על המידע עקבי. לנסות לממש את כל עקרונות CAP באיזשהו מובן, עם ביצועים מהירים ועקביים.



<u>מרכיבי הארכיטקטורה:</u>

1. **Batch Layer:** מנהלת את הנתונים ההיסטוריים של האפליקציה ומחשבת מראש batch views עבור שכבת ה-serving.
2. **Serving Layer:** פועלת על גבי שכבת הBatch. מוכנה לענות על שאילתות. 2 תפקידים עיקריים:
   a. מייצרת אינדקס על המידע ששמור על הmaster dataset (המידע ההיסטורי). כל פעם מבצעת עדכונים לאינדוקסים של המידע ב-master dataset (כל תקופת זמן שנגדיר במערכת).
   b. מייצרת view לmaster dataset שרלוונטיים לשאילתות נפוצות במערכת.
3. **Speed Layer:** מעכלת מידע נכנס ושומרת מידע מפרק הזמן האחרון (לפי איך שנגדיר), שכבה זו מייצרת view למידע ששמור בשכבה זו (מידע עדכני) שמתאים לשאילתות נפוצות.

רעיונות מרכזיים:
- speed layer – אחראית רק על מידע מפרק הזמן האחרון.
- שאילתה נוגעת לשתי השכבות speed ו-serving.

כשמגיעה שאילתה חדשה למערכת, אם אחת השכבות יכולה לספק לה מענה, אז יינתן לה מענה מאוד מהיר. אחרת, כל שכבה תיתן view שלה ונעשה איחוד של המידע. כך נוכל לתת מענה עבור שאילתות של real time וגם של הנתונים ההיסטוריים.

* עבור שכבת הbatch וה-serving נוכל להשתמש בHadoop, אבל עבור שכבת הspeed לא נוכל להשתמש בזה ונצטרך את Spark.

<u>לסיכום, ההבדלים בין map-reduce לRDBMS מקביליים:</u>

סגנון map-reduce – מצטיין בחישובים מורכבים ומשימות ETL (extract, transform, load).

DB מקביליים – מצטיינים בשאילתות על data sets גדולים.

שתי הגישות משלימות אחת את השנייה!