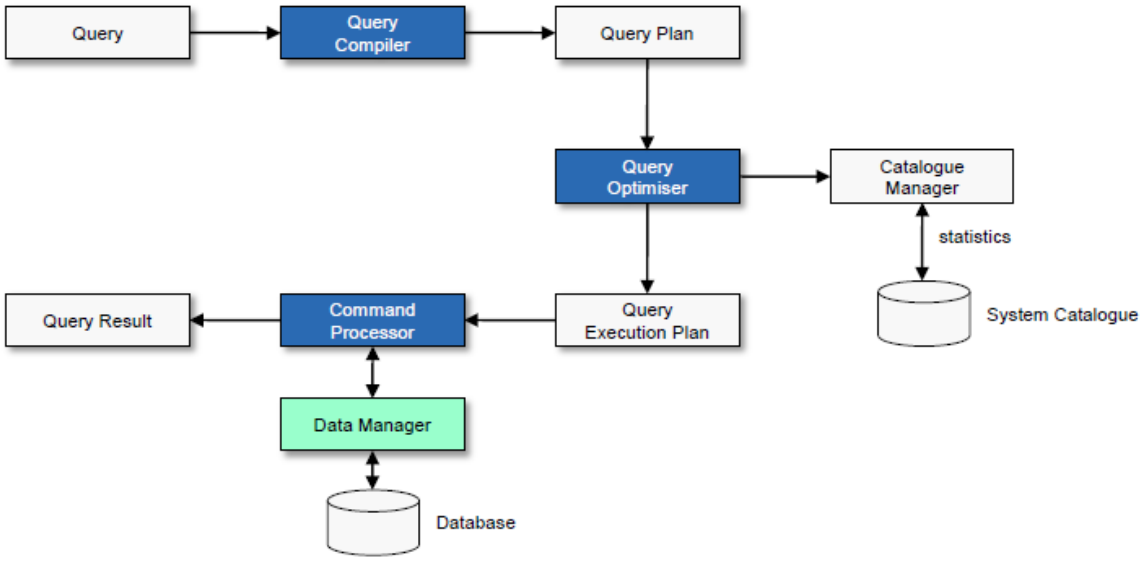
Przetwarzanie i optymalizacja zapytań

Generalny flow podczas przetwarzania zapytania:



Podstawowe kroki podczas przetwarzania zapytań:

1. Parsowanie i translacja zapytań (poziom kompilatora zapytań)
- weryfikacja składni
- weryfikacja istniejących relacji
- transformacja zapytania SQL do planu zapytania reprezentowanego w formie algebry relacyjnej (w przypadku relacyjnej bazy danych); warto wspomnieć, że możliwych postaci algebraicznych dla jednego zapytania jest więcej niż jedna

2. Optymalizacja zapytania (poziom optymalizatora)
- transformacja początkowego planu zapytania do najlepszego możliwego planu wykonania dla danego zestawu danych
- zaczynamy od specyfikacji wykonania pojedynczych składowych zapytania, kończąc na zdefiniowaniu sekwencji ich wykonania

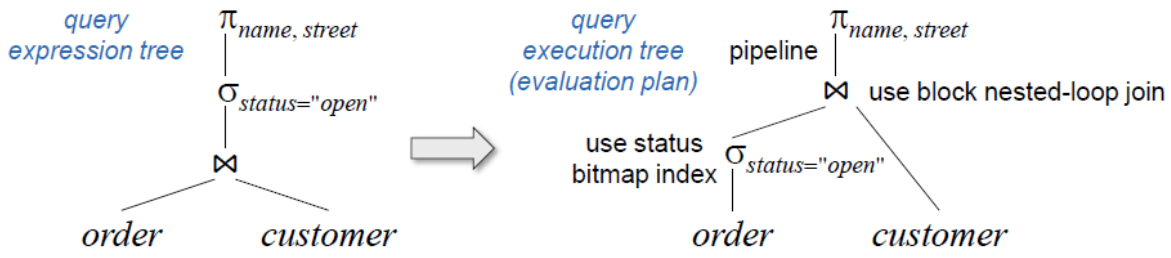
3. Wykonanie określonego planu z poziomu procesora.

Przykład:

SELECT *name, street*
FROM *Customer, Order*
WHERE *Order.customerID* = *Customer.customerID* AND *status* = 'open';

Transformacja do planu zapytania w formie algebry:

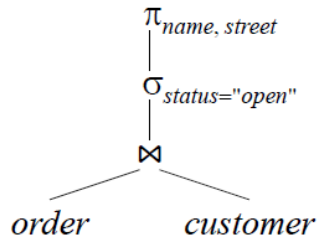
$$\pi_{name,\,street}(\sigma_{status="open"}(order \bowtie customer))$$



Istnieją dwa podejścia do wykonania drzewa zapytania:

- materializacja - wyliczamy wynik składowej zapytania i zapisujemy go jako nową relację (zmaterializowaną) na dysku
- pipelining - przekazywanie krotek do operacji nadrzędnej, nawet jeśli operacja jest wciąż wykonywana

Materializacja
Szacowanie operacji odbywa się po kolei, zaczynając od liści drzewa zapytania. Pośrednie wyniki materializowane są do tymczasowych relacji. Te wędrują do operacji na wyższym (następnym) poziomie drzewa. Koszt czytania i zapisywania tymczasowych relacji może być dosyć wysoki.



Wylicz $order \bowtie customer$ i zachowaj jako relację.
Wylicz $\sigma_{status="open"}$ na zmaterializowanej relacji i zachowaj.
Wylicz $\pi_{name, street}$ na zmaterializowanej relacji.

Pipelining
Szacuje wielu operacji jednocześnie przesyłając wyniki jednej operacji do następnej bez zachowania takich krotek na dysku. Ma to odzwierciedlenie w porównaniu do poprzedniego podejścia, ponieważ widać tutaj wyraźnie zaoszczędzenie na odczytach IO (poprzednio sporo na tymczasowych relacjach).
Występuje w formach:

- leniwy (lazy pipelining) – operacja z najwyższego poziomu w sposób ciągły odpytuje o nowe krotki od swoich dzieci
- gorliwy (eager pipelining) – dzieci produkują krotki dla swoich rodziców i wysyłają je do bufora; gdy ten jest pełny, dzieci czekają aż rodzic skonsumuje otrzymane krotki i zwolni miejsce

Przy optymalizacji zapytań można wyodrębnić alternatywne podejścia:

- inny, ale równoważny zapis zapytania (jego drzewa)
- inne algorytmy przy poszczególnych operacjach w drzewie zapytania

Nieco informacji odnośnie optymalizatora zapytań i kosztów, które są pewnego rodzaju wyznacznikiem w kwestii transformacji zapytań w celu ich możliwie najlepszego wykonania:

Optymalizator kosztowy opierając się o statystyki i dostępne struktury danych wytwarza plany wykonania zapytania. Bierze pod uwagę dostępne indeksy i obmyśla kilka możliwości dostępu do danych. Optymalizator kosztowy określi koszt dla każdego planu wykonania i wybierze ten plan, którego koszt będzie najniższy.

Optymalizator kosztowy określa podczas opracowywania planów m.in.:

- Koszt – wartość, w którą wliczane są cykle procesora czy ilość odczytów IO,
- Selektywność – ilość wierszy pobranych/całkowita ilość wierszy
- Liczebność – oczekiwana liczba wierszy (w wyniku zapytania) – selektywność*całkowitaIlośćWierszy.

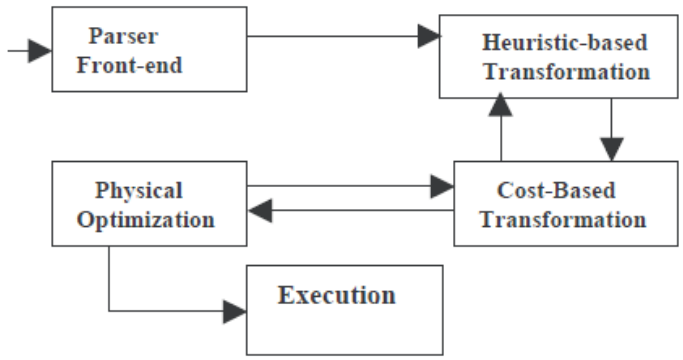
Czym jest koszt?
Metryczny **koszt** zgłaszany przez optymalizator jest wartością trudną do wyjaśnienia. Jest miarą złożoną, na którą wpływ ma:

- wartość szacowanych odczytów zarówno pojedynczych jak i mnogich bloków
- oraz szacowany czas CPU.

Wynik sumy tych składowych jest dzielony przez czas odczytu pojedynczego bloku danych. Teoretycznie mnożąc koszt przez czas odczytu pojedynczego bloku, możemy się spodziewać oczekiwanego czasu trwania zapytania w ms. Jednak będzie to rozczarowujące, ponieważ jak się okazuje na wspomniany czas wpływa jeszcze wiele innych czynników, w dużej mierze owianych tajemnicą. Tak więc sam koszt będzie bardziej użyteczny przy porównywaniu planów wykonania, aniżeli doszukiwaniu się czasów trwania. To wszystko oczywiście zakładając podobne warunki środowiskowe.

Koszty wyliczania kosztów
Przy transformacjach kosztowych warto również wspomnieć o konsumowanych zasobach - zarówno czasowych jak i pamięciowych. W Oracle struktury zapytań, decyzje optymalizatora i szacowane koszty, jako zasoby, nie są zwalniane dopóki optymalizacja nie dobiegnie końca. Framework do tworzenia transformacji kosztowych tworzy wiele kopii struktur zapytań oraz wiele wariantów zoptymalizowanych transformacji. Gdy operacja optymalizatora dotyczące danego zapytania zostaną zakończone zajęte zasoby pamięciowe zostają zwolnione. Oczywiście nie całkowicie, ponieważ wybrany plan może zostać ponownie użyty.

Transformacje zapytań w bazach Oracle



Obecne systemy oparte na relacyjnych bazach danych przetwarzają różnorakie skomplikowane zapytania SQL zawierające zagnieżdżone podzapytania z funkcjami agregującymi, operacje UNION, DISTINCT czy GROUP BY. Szybkość wykonania takich zapytań ma coraz większe znaczenie.

Oracle przeprowadza mnóstwo transformacji zapytań. Można wyodrębnić dwie grupy takich działań: heurystyczne i kosztowe. Należą one do fazy logicznej. Po niej następuje fizyczna, która obejmuje metody dostępu czy kolejność joinów.
Od wersji Oracle 10.1 wprowadzono transformacje oparte na kosztach. W wersjach poprzednich wykonywane były tylko transformacje heurystyczne. Oczywiście miało to wpływ na skomplikowanie działań, ale także na ich efektywność. Z pewnością wydłużył się czas parsowania zapytania. Ponadto transformacja akceptowana w Oracle 9i wraz z pojawieniem się transformacji opartych o koszty mogła zostać odrzucona właśnie ze względu na koszty wykonania.
Przykładowo w 9i zagnieżdżeń w zapytaniach próbowano się pozbyć - o ile było to możliwe. Od wersji 10g sprawdzane są koszty wykonywań zapytania z zagnieżdżeniem i bez niego. Wybierany jest plan z mniejszym kosztem.

Przykładowe transformacje zapytań

a.) Rozkład złączeń

```
SELECT e.first_name, e.last_name, job_id,
        d.department_name, l.city
FROM employees e, departments d,
        locations l
WHERE e.dept_id=d.dept_id and d.location_id=l.location_id
UNION ALL
SELECT e.first_name, e.last_name, j.job _id,
        d.department_name, l.city
FROM employees e, job_history j,
        departments d, locations l
WHERE e.emp_id=j.emp_id and j.dept_id=d.dept_id and d.location_id=l.location_id;

SELECT V.first_name, V.last_name, V.job_id, d.department_name, l.city
FROM departments d, locations l,
(SELECT e.first_name, e.last_name, e.job_id, e.dept_id
FROM employees e
UNION ALL
SELECT e.first_name, e.last_name,j.job_id, j.dept_id
FROM employees e, job_history j
WHERE e.emp_id=j.emp_id) V
WHERE d.dept_id=V.dept_id and d.location_id=l.location_id;
```

Rozkład Joinów (Join Factorization) wprowadzono w drugim wydaniu wersji 11g i dotyczy on zapytań z UNION ALL. Operator ten ma szczególne znaczenie np. przy integrowaniu danych. W wielu przypadkach części składowe zapytań z UNION ALL mają wspólną część procesową (np. w postaci odwoływania się do tych samych tabel). Jednak wspomniane składowe wykonywane są niezależnie, co prowadzi do powtarzalnego przetwarzania i dostępu do danych. Join factorization umożliwia wspólne wykorzystanie wyliczeń.
Analiza:
Patrząc najpierw na zapytanie czerwoną czcionką, później na to czcionką zieloną – można dostrzec, że złączenie po id lokacji (tabel departments i locations) odbywa się już tylko raz w tym przypadku poniżej. Można by to określić jako swoiste „wyciągnięcie przed nawias”. Reszta została zamieniona na widok, z którego wydobywane są pozostałe atrybuty. Widać, ze

budowa widoku składa się z tego samego union all, ale już bez złączenia departments i locations.

b.) Wyciągnięcie predykatu

```
SELECT *
FROM (SELECT document_id
FROM product_docs
WHERE contains(summary, 'optimizer',1)>0 and contains(full_text, 'execution', 2)
ORDER BY create_date) V
WHERE rownum < 20

SELECT *
FROM (SELECT document_id
FROM product_docs
WHERE contains(full_text, 'execution',2)>0
ORDER BY create_date) V
WHERE contains(summary, 'optimizer', 1)>0
AND rownum<20
```

Można powiedzieć, że predykat uznajemy jako kosztowny, gdy zawiera np. podzapytanie lub proceduralną funkcję (jak w naszym przypadku). W tej transformacji chodzi o to, by kosztowny predykat w formie filtru wyjąć z zawierającego go widoku do zapytania głównego zawierającego ten widok. Żeby to lepiej zobrazować popatrzmy na zapytania.
Jak widać na zamieszczonym slajdzie - drugi operator contains został wyjęty z widoku. Dzięki czemu może zadziałać na już zredukowanym zbiorze danych. Choć oczywiście nie zawsze tak musi być, bo czasem widok może być na dyle duży, że poprawa wydajności będzie niewielka.

c.) Rozwinięcie OR

```
Select *
From products
Where prod_category ='Photo' or prod_subcategory ='Camera Media';

Select *
From products
Where prod_subcategory ='Camera Media'
UNION ALL
Select *
From products
Where prod_category ='Photo'
And lnnvl(prod_subcategory ='Camera Media');
```

lnnvl(prod_subcategory ='Camera Media') – aby nie dołączać wierszy, które są z pierwszej części UNION ALL

Dotyczy zapytań zawierających operator OR. Główną ideą jest zastąpienie go poprzez transformację do postaci z UNION ALL, a wiec rozdzielenie na składowe - dwie lub więcej.

Dzięki rozwinięciu warunku OR na części możliwe stają się bardziej wydajne metody dostępu do tabel.
W pierwszym przypadku ze zwykłym ORem mamy do czynienia z dostępem to tabeli Products poprzez full scan, nawet pomimo faktu iż na obu kolumnach z warunku mamy założone indeksy. Dzieje się tak, ponieważ optymalizator przy operatorze OR traktuje dwa predykaty jako pojedynczą jednostkę. Po rozwinięciu ORa dostęp do tabeli Products odbywa się dwukrotnie, jednak z użyciem indeksu.
Warto dodać, że nie w każdym przypadku skorzystanie z indeksu będzie lepszym rozwiązaniem od full scana, jednak ta transformacja będzie wywoływana, gdy tak właśnie będzie.

d.) Ulokowanie group by

```
SELECT p.prod_id, sum(s.quantity_sold)
FROM Products p, Sales s
WHERE p.prod_id = s.prod_id
GROUP BY p.prod_id;

SELECT V.sumv, p.prod_id
FROM Products p,
(SELECT sum(s.quantity_sold) as sumv, s.prod_id
FROM Sales s
GROUP BY s.prod_id) V
WHERE V.prod_id = p.prod_id
```

Zmiana ulokowania operatora group-by umożliwia takie przekształcenie zapytania, by operacja grupowania odbyła się przed złączeniem.
Jak widzimy po transformacji operacja grupowania odbywa się wewnątrz widoku, co redukuje liczbę wierszy, które muszą być procesowane w trakcie hash joina pomiędzy SALES i PRODUCTS.
Skoro liczba wierszy użytych w złączeniu uległa zmniejszeniu – wydajność całego zapytania wzrosła (mniej wierszy do processingu przy group by).