

Sprawozdanie z laboratorium BD: 10, 11, 12

Indeksy, hinty

1. Indeksy - scan

Tabela WAREHOUSE – zawiera spis wszystkich magazynów magazynach:

Atrybuty: WAREHOUSE_ID (klucz główny), CITY (VARCHAR 25 bajtów),
CAPACITY(INTEGER), STREET(VARCHAR 500 bajtów)

Tablicę wypełniono 1000 rekordów, losowymi wartościami. Atrybut CAPACITY przyjmuje wartości od 10 do 1000. Atrybut STREET wypełniono stringami o długości 400 bajtów w celu zapewnienia dysku.

Zapytania:

- `SELECT * FROM WAREHOUSE WHERE CAPACITY <200;`

Selektywność takiego zapytania jest rzędu 20%

Wyniki bez użycia indeksu na CAPACITY: 66 (niezależne od selektywności)

Wyniki z użyciem indeksu na CAPACITY:

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT				95
TABLE ACCESS	WAREHOUSE	BY INDEX ROWID		95
INDEX	WAREHOUSE_CAPACITY	RANGE SCAN		2
Access Predicates				
CAPACITY <200				

- `SELECT * FROM WAREHOUSE WHERE CAPACITY <20;`

Selektywność takiego zapytania jest rzędu 1%.

Wyniki bez użycia indeksu na CAPACITY:

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT				5
TABLE ACCESS	WAREHOUSE	FULL		5
Filter Predicates				
CAPACITY <20				
Other XML				
				COST=66

Wyniki z użyciem indeksu na CAPACITY:

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT				5
TABLE ACCESS	WAREHOUSE	BY INDEX ROWID		5
INDEX	WAREHOUSE_CAPACITY	RANGE SCAN		2
Access Predicates				
CAPACITY <20				

- `SELECT * FROM WAREHOUSE WHERE CAPACITY=100;` (selektywność 0.1%)

Wyniki z użyciem indeksu na CAPACITY:

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT				2
TABLE ACCESS	WAREHOUSE	BY INDEX ROWID		2
INDEX	WAREHOUSE_CAPACITY	RANGE SCAN		2

Tabela ARTICLE – spis wszystkich dostępnych artykułów

Atrybuty: ARTICLE_ID (klucz główny), ART_NAME (VARCHAR 50 bajtów), CATEGORY (VARCHAR 1000 bajtów), VALUE (INTEGER)

Tablicę wypełniono liczbą 10000 rekordów, losowymi wartościami. Atrybut VALUE przyjmuje wartości od 10 do 1000. Atrybut CATEGORY wypełniono stringami o długości 800 bajtów w celu zapełnienia dysku.

Zapytania:

- `SELECT * FROM ARTICLE WHERE VALUE<200;`

Selektywność takiego zapytania jest rzędu 20%

Bez indeksu na VALUE:

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT				966
TABLE ACCESS	ARTICLE	FULL		6238
Filter Predicates				6238
VALUE<2000				

Z użyciem indeksu:

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT				603
TABLE ACCESS	ARTICLE	BY INDEX ROWID		596
INDEX	ARTICLE_VALUE	RANGE SCAN		596

- `SELECT * FROM ARTICLE WHERE VALUE<20;`

Selektywność takiego zapytania jest rzędu 1%

Bez indeksu na VALUE: uzyskano taki sam koszt jak poprzednio

Z użyciem indeksu:

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT				34
TABLE ACCESS	ARTICLE	BY INDEX ROWID		31
INDEX	ARTICLE_VALUE	RANGE SCAN		31
Access Predicates				31
VALUE<20				2

- `Select * FROM ARTICLE WHERE VALUE=100;`

Selektywność takiego zapytania jest rzędu 0.1%

Bez użycia indeksu na VALUE: uzyskano taki samo koszt jak poprzednio

Z użyciem indeksu:

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT				12
TABLE ACCESS	ARTICLE	BY INDEX ROWID		10
INDEX	ARTICLE_VALUE	RANGE SCAN		10
Access Predicates				10
VALUE=100				1

Posumowanie

W celu wymuszenia użycia indeksów zastosowano hinty: `SELECT /*+ INDEX(tabName) */ * ...`

W celu wymuszenia pełnego skanu użyto: `SELECT /*+ FULL(tabName) */ * ...`

Wyniki dla zastosowania `SELECT /*+ INDEX_FFS(tabName) */ * ...` będą przedstawione bezpośrednio w tabeli

WAREHOUSE

Selektywność	0.1%	1%	20%
FULL	66	66	66
INDEX	3	8	98
INDEX_FFS	3	8	66

ARTICLE

Selektywność	0.1%	1%	20%
FULL	966	966	966
INDEX	12	34	603
INDEX_FFS	12	34	603

Wnioski

- dla tabeli WAREHOUSE pokazano, że wymuszenie użycia indeksu spowodowało pogorszenie kosztów dostępu (aż o 50%) – optymalizator zapytań ORACLE nie używał indeksu, został on wymuszony
- w analogicznej sytuacji dla tabeli ARTICLE udało się obniżyć koszt zapytania o ok. 35%, jak widać pogorszenie nie jest regułą
- niezależenie od selektywności, koszt zapytania bez użycia indeksu pozostaje stały
- dzięki użyciu indeksu, wraz ze spadkiem selektywności, koszt zapytania znacząco maleje
- proporcja kosztu zapytania bez użycia indeksu do analogicznego z zastosowaniem indeksu (czyli de facto proporcjonalny zysk) znacząco rośnie wraz ze spadkiem selektywności
- użycie indeksów ma najwięcej sensu dla zapytań o selektywności <10%

2. Indeksy – join

Zapytanie 1.

```
SELECT * FROM WAREHOUSE INNER JOIN WAREHOUSE_ARTICLE ON  
WAREHOUSE_ID=WAREHOUSE_WAREHOUSE_ID;
```

Tabela WAREHOUSE_ARTICLE zawiera 12000 rekordów. Przechowuje ona dane odnośnie artykułów przechowywanych w magazynach WAREHOUSE. W każdym magazynie znajduje się średnio 10 artykułów.

1. Nested loop join – jako hinta użyto /*+ USE_NL(WAREHOUSE, WAREHOUSE_ARTICLE) */

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			12014	11068
NESTED LOOPS				
NESTED LOOPS			12014	11068
TABLE ACCESS	WAREHOUSE	FULL	1000	66
INDEX	WAREHOUSE_ARTICLE_PK	RANGE SCAN	12	1
Access Predicates				
		WAREHOUSE_ID=WAREHOUSE_WAREHOUSE_ID		
TABLE ACCESS	WAREHOUSE_ARTICLE	BY INDEX ROWID	12	11

2. Sort-merge join - jako hinta użyto /*+ USE_MERGE(WAREHOUSE, WAREHOUSE_ARTICLE) */

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			12014	298
MERGE JOIN			12014	298
SORT		JOIN	1000	67
TABLE ACCESS	WAREHOUSE	FULL	1000	66
SORT		JOIN	12014	231
Access Predicates				
		WAREHOUSE_ID=WAREHOUSE_WAREHOUSE_ID		
Filter Predicates				
		WAREHOUSE_ID=WAREHOUSE_WAREHOUSE_ID		
TABLE ACCESS	WAREHOUSE_ARTICLE	FULL	12014	229

3. Hash join - jako hinta użyto /*+ USE_MERGE(WAREHOUSE, WAREHOUSE_ARTICLE) */

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			12014	296
HASH JOIN			12014	296
Access Predicates				
		WAREHOUSE_ID=WAREHOUSE_WAREHOUSE_ID		
TABLE ACCESS	WAREHOUSE_ARTICLE	FULL	12014	229
TABLE ACCESS	WAREHOUSE	FULL	1000	66

Zapytanie 2.

SELECT * FROM ARTICLE INNER JOIN WAREHOUSE_ARTICLE ON
ARTICLE_ID=ARTICLE_ARTICLE_ID WHERE VALUE<500;

1. Nested loop join - jako hinta użyto /*+ USE_NL(ARTICLE, WAREHOUSE_ARTICLE) */

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			2626	12246
NESTED LOOPS				
TABLE ACCESS	WAREHOUSE_ARTICLE	FULL	2626	12246
INDEX	ARTICLE_PK	UNIQUE SCAN	12014	229
Access Predicates			1	0
	ARTICLE_ID=ARTICLE_ARTICLE_ID			
TABLE ACCESS	ARTICLE	BY INDEX ROWID	1	1

2. Sort-merge join - jako hinta użyto /*+ USE_MERGE(ARTICLE, WAREHOUSE_ARTICLE) */

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			2626	1470
MERGE JOIN			2626	1470
SORT		JOIN	1536	1239
TABLE ACCESS	ARTICLE	FULL	1536	966
Filter Predicates				
	ARTICLE.VALUE<500			
SORT		JOIN	12014	231

3. Hash join - jako hinta użyto /*+ USE_MERGE(ARTICLE, WAREHOUSE_ARTICLE) */

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			2626	1196
HASH JOIN			2626	1196
Access Predicates				
	ARTICLE_ID=ARTICLE_ARTICLE_ID			
TABLE ACCESS	WAREHOUSE_ARTICLE	FULL	12014	229
TABLE ACCESS	ARTICLE	FULL	1536	966
Filter Predicates				
	ARTICLE.VALUE<500			

Zapytanie 3.

SELECT VEHICLE.CAR_BRAND, VEHICLE.CAR_MODEL, VEHICLE.REGISTRATION_NUMBER,
VEHICLE.DATE_OF_PRODUCTION, REPAIR.DATE_OF_START, REPAIR.DATE_OF_END,
REPAIR.VALUE FROM REPAIR INNER JOIN VEHICLE ON
REPAIR.VEHICLE_ID=VEHICLE.VEHICLE_ID

Tabele VEHICLE i REPAIR mają po kilka rekordów. Na przykładzie takie złączenia pokazany zostanie sens użycia w niektórych przypadkach algorytmu nested loop join.

1. Nested loop join

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			8	4
NESTED LOOPS			8	4
TABLE ACCESS	VEHICLE	FULL	3	2
TABLE ACCESS	REPAIR	FULL	3	1

2. Sort-merge join

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			8	5
MERGE JOIN			8	5
TABLE ACCESS	VEHICLE	BY INDEX ROWID	3	2
INDEX	VEHICLE_PK	FULL SCAN	3	1
SORT		JOIN	8	3

3. Hash join

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			8	5
HASH JOIN			8	5
Access Predicates		REPAIR.VEHICLE_ID=VEHICLE.VEHICLE_ID		
TABLE ACCESS	VEHICLE	FULL	3	2
TABLE ACCESS	REPAIR	FULL	8	2

Podsumowanie

	Zapytanie 1.	Zapytanie 2.	Zapytanie 3.
Nested loop join	11068	12246	4
Sort-merge join	298	1470	5
Hash join	296	1196	5

Wnioski

- w zapytaniach 1. i 2. najgorsze wyniki uzyskano używając algorytmu nested loop join.
- w zapytaniach 1. i 2. najlepsze wyniki uzyskano stosując algorytm hash join.
- algorytm nested loop join jest zdecydowanie najwolniejszy dla dużych danych, ponieważ nie korzysta z żadnych informacji o porównywanych atrybutach – są to po prostu dwie zagnieżdżone pętle
- algorytm sort-merge join działa na dwóch posortowanych atrybutach, łącząc je w pętli; dzięki temu, że warunek łączenia założony był na atrybutach klucza głównego (które domyślnie posiadają index) łączenie tym algorytmem okazało się niewiele gorsze od hash join.
- algorytm hash join jest wybierany domyślnie przez optymalizator zapytań i sprawdza się również w wypadku, gdy dane nie są posortowane.
- algorytm nested loop join ma sens użycia jedynie w przypadku małej liczby danych, co pokazano na przykładzie zapytania 3. z małą liczbą danych
- dla zapytań operujących na małych zasobach danych względna różnica algorytmów jest niewielka